Sentry Security Camera

6.111 Final Project

Ray Wu, Bo Zhu, and Robert Speaker

14th December 2005
Section 1
Project Abstract

Safety is an important issue to many. Security personnel need to make sure the buildings they are guarding are well monitored. Homeowners and storekeepers that leave their property want to keep intruders away, or at least capture criminals on tape for the police. Because hiring guards and watchmen is expensive, a cheaper alternative lies in using today’s video and audio processing technology to do the job.

By using field programmable gate arrays (FPGAs), off the shelf circuit components, microphones, and a common digital video camera, we built a device to provide for the aforementioned security needs. The Sentry Camera is designed to provide a security monitoring machine that uses visual and audio input to identify a target that’s either moving or producing noise. This device can be installed in homes and facilities to monitor the premise for intruders. Both motion and sound that trigger the device can then be used as signals to perform a host of responses, including tracking the target and recording it.
Section 2
Table of Contents

1. Abstract 1
2. Table of Contents 2
3. List of Figures 3
4. List of Tables 4
5. List of Hardware used 5
6. Overview 6
7. Module Descriptions 8
   a. Robert’s Modules:
      i. Master Controller 8
      ii. Motion Calculator 11
      iii. Motor Control 15
      iv. Display Module 19
      v. Picture Taker 21
      vi. Test angle generator 22
   b. Ray’s Modules
      i. Video Processor Overview 23
      ii. NTSC To RAM 25
      iii. VGA With RAM 27
      iv. CalcMBError 29
      v. Calc_camera_angle 30
      vi. Video RAM 31
      vii. Video Testing and debugging 32
   c. Bo’s Modules
      i. Audio Processor Overview 33
      ii. Microphones Amplifiers and ADCs 34
      iii. ADC Controller 35
      iv. Sound Source Location 39
         Algorithm
      v. Differencer 43
      vi. Slopefinder 45
      vii. Angle Calculator 46
      viii. Testing and Debugging 47
8. Conclusion 48
### Section 3

#### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Overall Block Diagram of the whole system showing all the subcomponents and how they interact with each other.</td>
<td>6</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Block diagram of the Motion Calculator module</td>
<td>11</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Angles are represented as a 6 bit number from 0-35 giving precision within 10 degrees.</td>
<td>12</td>
</tr>
<tr>
<td>Figure 4</td>
<td>The layout of a Unipolar (Bifilar) motor and the four coils used to drive it (IA+, IA-, IB+, and IB-)</td>
<td>15</td>
</tr>
<tr>
<td>Figure 5</td>
<td>A Block diagram showing the wiring of the motor to the driver chip</td>
<td>17</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Overall block diagram of the video processing module</td>
<td>23</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Video Recording timing</td>
<td>25</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Video Interframe comparison</td>
<td>26</td>
</tr>
<tr>
<td>Figure 9</td>
<td>VGA parameters for various resolution and frame rates</td>
<td>27</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Macroblock comparison description</td>
<td>28</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Calculation of Video Angle</td>
<td>29</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Block diagram of the Audio Processing Module.</td>
<td>33</td>
</tr>
<tr>
<td>Figure 13</td>
<td>The microphone amplifier and AD7871 ADC setup</td>
<td>34</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Timing Diagram for AD7871 Analog to Digital Converter.</td>
<td>35</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Logic Analyzer output for ADC Controller.</td>
<td>37</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Finite State Machine of the ADC Controller.</td>
<td>37</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Triangle model of Time Difference of Arrival</td>
<td>39</td>
</tr>
<tr>
<td>Figure 18</td>
<td>The possible locations for the sound source</td>
<td>41</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Superposition of audio vectors</td>
<td>42</td>
</tr>
</tbody>
</table>
Section 4
List of Tables

Table 1
A mapping of each switch on the labkit and its corresponding function 8

Table 2
A mapping of each button on the labkit and its corresponding function 9

Table 3
State transition diagram to drive the Unipolar stepper motor 16

Table 4
Timing Characteristics of AD7871 Analog to Digital Converter 35
Section 5
List of Hardware Used:

1. AIRPAX LB82731-M1 Stepper Motor
2. LM18293N Push-Pull Driver Chip
3. Camera Mount
4. 2 275-017A SPDT Submini Roller Lever Switches
5. 4 Omni-directional Condenser Microphones
6. 4 External AD7871 ADC’s
7. CCX-Z11 Video Camera
8. External Power supply for motors
9. 6.111 FPGA labkit
Section 6: Overview

This document outlines the sentry security camera design project that we created for our 6.111 final project. In this project we came up with a system that is capable of tracking motion and centering the camera on that motion. In addition the system is capable of sensing when a loud crash or bang occurs somewhere outside the field of view of the camera and will move the camera to locate where the sound is coming from. If someone were breaking into a store upon breaking the glass the camera would detect this sound, move the camera to the location of the sound and begin visual tracking of the target. The camera will follow the target wherever it detects motion. The system will also take pictures of the motion it detects. To implement this design we broke up the system into parts. Ray was responsible for the Visual motion tracking. Bo was responsible for the audio detection and location. And Robert was responsible for the motor control, video capturing, and integration of the other components. An overall block diagram of the system can be seen in figure 1.

![Figure 1: Overall Block Diagram of the whole system showing all the subcomponents and how they interact with each other.](image)

As with any large project the best way to go about working on it is to break the large problem into smaller sub problems. The block diagram above shows the smaller modules that piece together to form the system as a whole. The Master Controller is the integrating module that connects all the smaller modules together. It is responsible for
managing all the user input. The Video Processor is responsible for taking the camera input and detecting motion using a sum of absolute luminance differences algorithm. Similarly, the Audio Processor is responsible for taking input from the microphones and calculating where the sound came from by comparing the time delay between each audio signal. The Master Controller then feeds these results to the Motion Calculator which determines which direction it needs to move the motor to center the camera on the motion/audio. In addition to this, the Master Controller sends debugging information to a Display Module that displays this information on the LCD screen. Each one of these modules will now be described in detail.
Section 7.a.i Master Controller (Robert)

The master controller is responsible for integrating all the sub modules together and connecting everything up to the FPGA’s input and output connections. It is based off of the template labkit.v file written by Nathan Ickes. It incorporates some modifications used to set up the XVGA and ZBT memory (which will be discussed in section 7.a.v) written by Javier Castro and Ike Chuang respectively. One of the main jobs of the master controller was to take the input from the labkit’s buttons and switches and feed them to their appropriate modules. For more information on what each switch/button does you can look at the section marked in the Section Reference column.

Table 1: A mapping of each switch on the labkit and its corresponding function

<table>
<thead>
<tr>
<th>Switch Number</th>
<th>Name</th>
<th>Brief Description</th>
<th>High</th>
<th>Low</th>
<th>Section Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Manual Override</td>
<td>Toggles Between Automatic Motion and Manual</td>
<td>Manual Motion</td>
<td>Automatic Motion</td>
<td>7.a.ii</td>
</tr>
<tr>
<td>1</td>
<td>Debug Angles</td>
<td>Switches between the angles generated from video and audio processors and debug angles generated for testing</td>
<td>Generated Debug Angles</td>
<td>Real Angles</td>
<td>7.a.vi</td>
</tr>
<tr>
<td>2</td>
<td>Unused</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>Continuous Video Feed</td>
<td>Switches between active comparison on two video frames and comparison of one active and one still background</td>
<td>Discontinuous-Still Background</td>
<td>Continuous-Two Active Frames</td>
<td>7.b</td>
</tr>
<tr>
<td>4</td>
<td>High Contrast Video Mode</td>
<td>Toggles the video contrast for better detection by increasing the false positive rate</td>
<td>High Contrast</td>
<td>Low Contrast</td>
<td>7.b</td>
</tr>
<tr>
<td>5</td>
<td>A/V Priority</td>
<td>Tells the Motion Calculator what takes priority, visual motion or audio detection</td>
<td>Audio trumps video</td>
<td>Video trumps audio</td>
<td>7.a.ii</td>
</tr>
<tr>
<td>6</td>
<td>Video Processing Display</td>
<td>Controls whether the interframe comparison is displayed on the LCD or not</td>
<td>Display interframe video comparison</td>
<td>No display</td>
<td>7.b</td>
</tr>
<tr>
<td>7</td>
<td>Picture Viewer Display</td>
<td>Controls whether the display module or the picture view is displayed on the LCD</td>
<td>Picture Viewer Module</td>
<td>Display Module</td>
<td>7.a.iv</td>
</tr>
<tr>
<td>Button</td>
<td>Name</td>
<td>Description</td>
<td>Section Reference</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enter</td>
<td>Reset</td>
<td>Re-initializes all modules</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Snapshot</td>
<td>When in Manual Override mode will take pictures of the current video display</td>
<td>7.a.v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Unused</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Simulated Audio</td>
<td>When in Debug Angle mode will trigger the motors to move as a result of simulated audio motion</td>
<td>7.a.ii/7.a.vi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Simulated Video</td>
<td>When in Debug Angle mode will trigger the motors to move as a result of simulated video motion</td>
<td>7.a.ii/7.a.vi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Up</td>
<td>Move up</td>
<td>When in Manual Override mode will move the camera up</td>
<td>7.a.ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Down</td>
<td>Move down</td>
<td>When in Manual Override mode will move the camera down</td>
<td>7.a.ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Left</td>
<td>Move left</td>
<td>When in Manual Override mode will move the camera left</td>
<td>7.a.ii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Right</td>
<td>Move right</td>
<td>When in Manual Override mode will move the camera right</td>
<td>7.a.ii</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The master controller takes the inputs from the switches and buttons listed in the tables above and will feed them to the appropriate modules that do the processing. All of these buttons and switches are debounced and synchronized with a 65 MHz clock that the master controller generates. The debouncing is necessary to remove any intermediate noise that is generated from manual switches. The synchronization makes integrating all the modules much simpler since everything is running on the same clock. The reason that a faster clock than the labkits normal 27 MHz clock was used was because in order to display 1024x768 resolution on the LCD (refreshing at 60 Hz) a much faster clock speed is needed. This increased clock speed made writing to the ZBT and video ram more difficult because of propagation delays. After interlacing the read and writes across different clock cycles everything could be driven at the faster 65 MHz clock.

The master controller also integrates the sub modules together. It takes the angles generated from audio and video processing units and feeds them to the motion calculator module. The motion calculator then uses the angles to determine how to move the motor. The master controller takes the video input from the camera via the composite video-in on the labkit and feeds these signals to the onboard video RAM and ZBT RAM. The video processing unit uses the video RAM to compare frames to find motion. The master controller stores the video into the ZBT RAM whenever the user wants to take a picture or whenever motion is detected (see section 7.a.v for details). The master controller also generates debugging information to display on the labkits LED’s.

Integrating components together is almost always more difficult than one anticipates and so we prepared for this early in order to get adequate debugging and testing time. Every so often in the development cycle we would share each others code.
to make sure that we knew what the other’s modules were doing to make sure that no
one interpreted specifications differently and that implementations were compatible.
This worked remarkably well because the integration was very straightforward and the
only real bugs we had were with spelling typos and a few problems with the video
display. Originally, the video processing unit was being driven at a 78 MHz clock in
order to draw frames at 75 Hz on the LCD but we had to downgrade this speed in order to
make the modules compatible. This was not a very large change since the LCD does not
care if the it is begin driven at 75 Hz (78 MHz clock) or 60 Hz (65 MHz clock). Testing
the picture taking mechanism was very easy as well because it was easy to see whether
the pictures were displayed on the screen or not. The only difficulty was in choosing the
parameter of how long to hold the write enable high. We decided to use 0.1 seconds
because through experimentation we found that was a reasonable value for catching a
person walking by the camera.
Section 7.a.ii Motion Calculator (Robert)

The Motion Calculator module is responsible for getting the information from the audio and video processors and determining which direction to move the camera in order to find this motion. A block diagram of the motion calculator can be seen in figure 2.

Figure 2: Block diagram of the Motion Calculator module

The Motion Calculator can run in two different modes depending on the state of the Manual Override switch. If Manual Override is asserted then the motion calculator will not listen to the video and audio angles that are given to it, but will instead take input from the user in order to determine which direction to move the camera. The m_move[3:0] signals come from the labkits up, down, left and right buttons and will move the camera in the specified direction. Using this mode was very useful for debugging the control of the motor before the audio and video processor code was complete. This also made it a lot easier to get the correct timing for stepping the motor coils as discussed in section 7.a.iii. When Manual Override is not asserted the Motion Calculator ignores the user inputs and runs in automatic mode. It takes the angles from the video and audio processors and determines where to move the camera. The signals v_angle_h[5:0] and v_angle_v[5:0] are the horizontal and vertical angles relative to the current view of the camera that the video processing unit has determined motion to be found at. The v_motion signals tells the Motion Calculator when these angles are valid, i.e. when motion has been detected. Simarly, a_angle[5:0] is the angle relative to the current camera view that the audio processing unit has determined sound to be coming from. a_motion is the signal when sound is detected and the angle has been computer. Since the motor can only move in 10 degree steps (see section 7.a.iii) we only need to report angles with 10 degree precision. We, therefore, represent the angles as a 6 bit
number from 0-35, where each number multiplied by 10 represents the actual angle degree. This can be described best by looking at figure 3 which shows the possible angle values relative to the camera view.

![Figure 3: Angles are represented as a 6 bit number from 0-35 giving precision within 10 degrees.](image)

If the motion calculator will determine whether to turn clockwise or counter clockwise based on whichever takes less steps. For example, if it is given an angle of 2, it will tell the motor control unit to move clockwise for 2 steps of the motor, whereas if it is given a 35 it will move 1 step counter-clockwise instead of 35 steps clockwise. A design decision was made to keep track of all angles relative to the current view of the camera. Keeping track of angles in this fashion drastically reduces the complication that would result if the motion calculator needed to keep a state of where it has already moved. If the motor was told to move 30 degrees but in actuality it only moved 29 then over time these off by one errors would add up and keeping track of the angle state would be infeasible. Furthermore, if relative angles were not used, there would be added complication due to the fact that the motion calculator would have to initialize itself. When the system is first powered on, the calculator has no idea in what direction the camera is facing so it would need to go through some initializing step to set the initial angle state. Keeping track of the relative angles meant that no initialization was needed and that there was no stored state so off by one angles have no effect other than being off by one. The downside of using relative angles was that everything must move with the camera, thus increasing the weight that the motor has to move. All of the microphones and wires required for the audio processor needed to be mounted on top of the motor, in addition to the camera and all of its wires. This added weight made it very difficult for the motors to produce enough torque to move the motors. Furthermore, the additional wires upon spinning of the motor often became twisted and limited the motor from spinning to its full capacity.

The motion calculator was implemented so that whenever the motor is currently moving it will not accept any inputs from the video or audio processing units. When the camera is moving it will always detect motion so we need to ignore this input. After movement has stopped the calculator continues to ignore input for a configurable amount of time (2 seconds was our final solution). This blanking period was necessary because when the motor stops moving there is inevitably some jitter from the motor coming to a
sudden stop that could be misinterpreted as motion. To prevent the video processor from sending the motor in a continuous stream of motion because of post-movement jitter, we decided to ignore signals for 2 seconds after movement has ceased. The downside of this decision is that the camera is slower to follow objects in motion because it takes 2 seconds after each movement before it begins processing again. Whenever we are in this delay state, the calculator sets the output blank high so that other modules know that it is not accepting any input to avoid unneeded computation.

The motion calculator takes in a priority signal that is used to determine which takes priority when both audio and video are detected at the same time. If priority is low then video will trump audio. If any audio signal comes in when we are already processing a video angle then the audio signal will be ignored. If priority is high then the opposite will happen. This was mainly used for debugging purposes to test each component of the system individually.

The motion calculator produces several outputs. It produces a moving signal that is set high whenever the motion calculator is driving the motors. This signal is sent to the video and audio processing units so that they can avoid doing any computation during times when the motion calculator would not listen to them anyway. The complements to this signal are the h_expire and v_expire signals which signal when movement has finished and the processing units can begin to start sending new angles. In the final implementation the video processing unit ignored these signals and computed a new angle regardless of whether the motor was moving or not. However, the audio processing unit was reset after movement terminated and began recalculation.

The other outputs of the motion calculator module are the signals sent to the motor control modules that actually drive the motor. The look_left, look_right, look_down, and look_up signals are the fwd/reverse signals sent to the horizontal and vertical motor controllers which drive the motors in the clockwise/counterclockwise direction as described in section 7.a.iii.

Originally, our design incorporated 2-dimensional movement. After experimenting with the motors we ran into difficulty getting movement along the y-axis because the stepper motors we chose (see section 7.a.iii) did not have enough steady-state torque to hold the camera still when the motor was not driven. The system still generates all the signals necessary to move the camera along the y-axis despite the fact that these signals were not actually connected to a motor in our final design. The cant_move_up and cant_move_down signals are inputs that tell the motion calculator when the motor cannot move anymore because of mounting constraints. Whenever the motor reached its maximum position it would hit a doorbell switch that prevented any further motion in this direction. The switches used for this were 275-017A SPDT Submini Roller Lever Switches running from an external 5V power supply and fed into the labkits user ports.

Testing the motion calculator module went smoothly. Using the test angle generator described in section 7.a.vi we were able to test the motion calculator module before the audio and video processing modules were completed. The test angles were sent to the motion calculator and we were able to see that the motor would move the specified number of degrees (within 10 degree increments). We also experimented with the timing of when the angles were asserted to make sure that the angles were ignored when the motor was in motion or we were in the blanking state. The display module
made debugging the motion calculator really easy and so this was one of the first modules to be completed.
Section 7.a.iii Motor Control (Robert)

The motor control module was responsible for driving the motor to move the camera. We chose to use stepper motors because their movement is much more controlled than a DC motor. Stepper motors are capable of moving a discrete distance at each step. Another possibility would be to use a servo motor which is also capable of moving a discrete angle (and possibly more accurately) but servos are more expensive than the simple stepper motors we used. We found stepper motors already in the lab room so we decided to use them since they were available and we could start using them immediately without having to wait for ordered parts to arrive. In hindsight, it might have been better to get more efficient motors, because the motors we used had difficulty producing enough torque to move the required weight. The specific type of stepper motor that we used was an AIRPAX LB82731-M1 Stepper Motor. These motors are unipolar (Bifilar), the layout of which can be seen in figure 4.

![Unipolar Motor Diagram](image)

**Figure 4: The layout of a Unipolar (Bifilar) motor and the four coils used to drive it (IA+, IA-, IB+, and IB-)**

The AIRPAX LB82731-M1 Stepper motors have 4-75 Ohm Coils that are used to step the motor in both full and half-step increments. We decided to use half-step increments because we needed more precision than that provided by full steps. From experimental testing of the motors (because the spec sheets were not available), the full steps moved the motor about 20 degrees compared to the 10 degrees for each half step. The motor control module is essentially an FSM that generates the IA+, IA-, IB+, and IB- signals that are sent to the motor to charge and discharge each of the 4 coils. The state transition diagram for half step increments can be seen in Table 3. The motor is capable
of moving in both clockwise and counterclockwise directions depending on which order
the states are traversed. There are 8 states all together and experimental testing showed
that for reasonable performance a maximum of a 10 Hz clock could be used to transition
from state to state without skipping any steps. The faster you transition between the
states the less torque the motor will have since it has less time to charge each coil. We
decided to use a 5Hz clock to allow the quick movement necessary to track visual motion
and still provide the necessary torque to move the camera mount.

Table 3: State transition diagram to drive the Unipolar stepper motor in half step
increments. Increasing state produces clockwise movement and decreasing state
produces counter-clockwise movement.

<table>
<thead>
<tr>
<th>State</th>
<th>IA+</th>
<th>IA-</th>
<th>IB+</th>
<th>IB-</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>On</td>
<td>Off</td>
<td>On</td>
<td>Off</td>
</tr>
<tr>
<td>2</td>
<td>On</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
</tr>
<tr>
<td>3</td>
<td>On</td>
<td>Off</td>
<td>Off</td>
<td>On</td>
</tr>
<tr>
<td>4</td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
<td>On</td>
</tr>
<tr>
<td>5</td>
<td>Off</td>
<td>On</td>
<td>Off</td>
<td>On</td>
</tr>
<tr>
<td>6</td>
<td>Off</td>
<td>On</td>
<td>Off</td>
<td>Off</td>
</tr>
<tr>
<td>7</td>
<td>Off</td>
<td>On</td>
<td>On</td>
<td>Off</td>
</tr>
<tr>
<td>8</td>
<td>Off</td>
<td>Off</td>
<td>On</td>
<td>Off</td>
</tr>
<tr>
<td>1</td>
<td>On</td>
<td>Off</td>
<td>On</td>
<td>Off</td>
</tr>
</tbody>
</table>

The motors require a lot of current to charge and discharge each coil. Even
though the labkit is capable of producing enough voltage to drive the stepper motor, we
decided to use an external power supply to avoid surges of current that may interfere with
other signals driven by the labkit. A 12V power supply was used to supply the current.
We also used a LM18293N Push-Pull Driver Chip to amplify the signals coming from the
labkit. A layout of how the motors are wired up to this chip can be seen in figure 5.
One of the biggest problems we had with the motor was that when the motor was not being driven, there was not enough torque to hold the camera mount in place. When there was motion, the motor could move the mount but when motion ceased the motor was not powerful enough to hold the vertical mount up. We had to remove vertical motion from the system because there was no way to get the mount to stay steady. Oscillating the motor back and forth would not work because the video processing unit would detect these oscillations as motion. Another downside of the fact that the motors had very low steady state torque was the interference with the microphone and camera wires. When we integrated all the components together, the motor would not always move in the desired direction since it kept being pulled back by the wires. If we were to do this project again we would probably make use of some other type of motor with more power to avoid this problem.

The motor was mounted on an aluminum casing that provides 360 degrees horizontal rotation. The original mount allowed vertical motion as well but this part was removed after we decided to limit the rotation to 1-dimension. The mount was also modified to support the microphones that needed to be 2 ft apart in order to get accurate audio signal delay times. To limit the weight of the mount we resourcefully recycled a cardboard pizza box to support the microphones. A picture of the mounting display can be seen below.
Testing the motor control consisted of using the Manual override mode to move the motor in the user specified directions. Getting the state transition timing proved to be very difficult without the data sheet for the motors. After some manual testing of connecting various motor wires to the power supply we were able to determine the appropriate state machine and wiring configuration. We experimented using the motor with varying weights to get a sense of how much it could handle. Unfortunately the amount of weight of all the microphones and the camera was just on the threshold of how much the motor could tolerate. We perhaps pushed the motor past its limits because on the day before the project presentation the original motor burnt out. Luckily, we had a spare motor that we were able to use as a replacement. Because of the large number of wires used by the microphones we ran into a lot of unforeseen problems with the motors interfering with the motion when we integrated all the components together. Eventually, we decided the best option was to coil all the wires up and hang them vertically so that they would twist with the motor. However, the added resistance from the wires was enough to pull the motor back and interfere with the motion. If we were to do this again we would probably get the help of some mechanical engineer to come up with a better wiring solution.
Section 7.a.iv Display Module (Robert)

The display module served as a debugging tool for both the video and audio processing units. It uses the XVGA module written by Javier Castro to display pixels to the LCD screen. A screen shot of the display module in action can be seen below.

As you can see from the picture the display module is broken up into 4 quadrants each displaying its own debugging information. Each quadrant will be explained separately.

Upper Left Quadrant:
The Upper Left Quadrant was a debugging tool for the audio processor that pictorially represented the vectors corresponding to the possible location of the audio source. There are 4 pink squares that represent the 4 microphones, which were arranged all on the same plane at 90 degree angles. Whenever sound is detected the horizontal and vertical pairs of microphones each act independently to find the possible direction that the sound is coming from. Each pair can limit the number of possible locations down to two vectors, meaning there are 4 to choose from. This quadrant is displaying the 4 different vectors (represented as white squares) and their relative distance from each microphone. The intersection of these vectors is the calculated location.
Upper Right Quadrant:
The Upper Right Quadrant was used as an extension to the information displayed in the upper left. Once the audio processing unit has limited the number of possible sources of audio down to 4 vectors it then finds the intersection of these vectors in order to find the actual location. The upper right quadrant displays this location as a pink square on an X-Y plane that represents the plane that the microphones share. Even though we only needed to determine the angle to find the motion we thought it would be useful to display the predicted location as well to get a feel for the distance of the object.

Lower Left Quadrant:
This quadrant was used to help debug the motion calculator unit to make sure it was telling the motor to move in the right direction, regardless of whether the motor was actually moving in that direction or not. A single circle with arrows depicting which direction the camera is trying to move is displayed in this quadrant. This was useful in the early stages when we were having difficulty getting the motor to move because it allowed us to continue work on the other modules and see that they were generating the correct signals even though the motor control module was not working.

Lower Right Quadrant:
In the lower right quadrant there are 3 compasses that display the angles that the video and audio processing units are saying they detected motion in. The angles are reported using the same format that they are reported to the motion calculator module (see figure 3). The white compasses represent the horizontal and vertical angles from the video processor and the red compass is the horizontal angle from the audio processing unit. The blue squares represent the angles with 0 degrees being straight up.

These visual debugging tools made it a lot easier to test all the components because it is much easier to see on a screen what the system is trying to do than it is to interpret signals on the logic analyzer. As you can see its not the prettiest interface because there is some noise on the display. This is a result of the circles taking too much computation time to do the $x^2 + y^2 \leq r^2$ computation. A better implementation would have used ROM’s to draw the circles, since they are in fixed positions. However, since this was just a debugging tool we decided it was not necessary to re-implement everything.
Section 7.a.v: Picture Taking (Robert)

Since we are marketing our device as a security camera we thought that it would be a good idea to record images in addition to just following the motion. It would be inefficient to store video 24 hours a day 7 days a week so we decided to simply capture screenshots using the labkit’s ZBT RAM. Pictures can be taken in one of two ways. One way is at the user’s control. When the system is running in Manual Override mode (see section 7.a.ii), a user can take a picture of whatever is currently in view of the camera by hitting button 0 on the labkit. The system also takes pictures automatically whenever video motion is detected. Upon assertion that the video processor has found motion the ZBT write enable is held high for a short duration, capturing a single frame onto the screen. Note that picture taking is handled within the master controller code and is not a standalone module.
Section 7.a.vi: Test Angle Generator (Robert)

One of the hardest parts about working in a group is managing your work schedule so that you are not blocked waiting for group members to complete their part of the assignment. Since I knew that it was going to take awhile for my group members to finish the audio and video processing units, I decided to implement my own test module that would simulate their functionality. The test angle generator module simply assigns values to the \texttt{a\_angle}, \texttt{v\_angle\_v}, and \texttt{v\_angle\_h} signals and increments every so often. This allowed me to test the automatic motor movement before the audio and video processors were complete and also made the integration easier since it was simply a matter of replacing the fake signals with the real ones.
We tried to make our device as real-world as possible. A sentry camera connected to a recording device or an alarm should accurately detect movement (intruders). However, while it needs to be sensitive, it should also minimize false alarms. In our case, we wanted to have the camera track motion and follow the moving object. Motion detection can theoretically set off a host of responses, from calling the police to recording the intruder on video. In our case, when the video processing module detects motion, the motion is recorded in zbt memory and the moving target is tracked by having the camera move toward the motion source.

The sentry camera receives continuous, raw video data from a Sony digital video camera. In order to process motion and address hardware limitations, however, the data is stored and modified in an efficient manner. In addition, a motion tracking algorithm with high success rate, capable of differentiating signal from noise in addition to tracking motion under various lighting conditions, had been developed.

The sony digital camera feeds a bitstream with at an approximately 14Mhz clock to an onboard ADV7185 chip. The chip sends digital video data from the camera to an ntsc_decode module to convert the data into a 24-bit stream of YCrCb (8 bits luminance and a pair of 8-bit chrominance). For the project, only the Y (luminance) values were used. The 8-bit Y data, along with the camera hsync, vsync, and data valid signals were sent to the video processing module.

The general block diagram for the video processing module is shown in Figure 6.
Figure 6: Overall block diagram of the video processing module. Luminance video data coming in from the NTSC_Decode module is assigned to video data registers and assigned to be stored in the appropriate locations in the VideoRam. The master Video_Processing module (vga_with_ram) interfaces with the ram and also sends luminance data to be processed for motion to Calc_MBErro. If motion is detected, Calc_Camera_Angle is used to find the real-world angle the camera needs to be moved to center on the motion.
Section 7.b.ii NTSC_TO_RAM (Ray):

Luma values are continually fed by the video camera and need to be stored synchronously to the videoram. This module takes the 8 bit luma values, and adjusts the videoram address (vaddr) and data (vdata) appropriately. Furthermore, because the videoram has memory limitations, the module samples every other pixel horizontally and every other line. NTSC_TO_RAM stores at 256x192 addresses, with each address having 8 bits, corresponding to the size of each pixel luma value. Addresses are 16 bits with the first 8 bits corresponding to row and the second corresponding to column.

Because both a history and current frame need to be stored and because motion capture should operate for slow motion, a two-frame memory architecture with a delay was added. The signal “record” goes high when the 60Hz fvh[2] (frame) goes high after the set number of frame delays. When record is high, either storeInA or storeInB is high, instructing the ram to store either in the memory allocated for frame A or that for frame B. The two store signals alternate, so that the two frames stored are always a set number of frames away from each other.

A timing example is shown in figure 7:

![Timing diagram](image.png)

**Figure 7:** Record goes high in this example every other frame, triggering on the posedge of fvh[2], (skipping every other frame). Memory write (vwe) is high whenever record and dv are high and fvh[2] is low. storeInA and storeInB alternate each time record occurs.

Each of the two frames takes up 256x96 addresses, and the module switches between storing in memory A and B, with an adjustable delay of frames in between (default is 4 frames). In order to switch from storing in frame A to storing in B, a shift of 96 is added to the “row” address of vaddr.

The camera also sends vsync and hsync signals, serving to align the \{row, col\} address. Every time vsync is received, both row and column addresses are reset to 0. hsync resets only column addresses to 0. For other data-valid signals (dv high), column is incremented until 256 column (and 96 row) address are filled. Because of limitations in the block ram memory size, not every pixel in a frame from the camera is stored. Instead, every fourth pixel on each line and every other line are stored. In the vga_with_ram
module, the process for displaying the pixel onto the screen reverses the sampling process, displaying every pixel in memory four times in a row and column.

Several video input adjustments are implemented in this module. In order to increase contrast in some situations, pixel luminance can be changed to be either black (8’d0) or white (8’d255) with a threshold of 8’d120 determined through trial and error. All luma values received that are higher than 120 will be converted to 255 and all values under the threshold are changed to 0. If the high contrast feature is turned off, the 8-bit luma values received are unchanged.

Another feature added to improve motion detection was to freeze frame storage in memory B while continuing to write new frames into memory A. As long as the camera does not move, the motion detection algorithm is comparing the background to whatever appears in the foreground (in frame A), significantly improving motion detection and serving as an enhancement tool with practical applications.

The outputs of the module go directly to the vga_with_ram module, which uses vaddr, vdata, vwe, and record to store the luminance data into appropriate either location A or B in memory. A display of the two memories outputting a history and a current frame is shown in figure 8.

Figure 8: Luminance data from memory A (top picture display) and B (bottom picture display) are shown above on the monitor. The two displays are a constant frame length apart in order to allow for motion detection to capture slower movement.
Section 7.b.iii VGA_WITH_RAM (Ray):

Video data, memory address location, and record signals sent from the ntsc_to_ram module are used by this module to write the video data into ram. A 65 MHz pixel clock is generated with the appropriate clock signals for displaying pixels on the computer monitor. Several different clock setups could have been chosen, but because the screen displayed data at 1024x768 pixel resolution, and we wanted to have a clock speed that’s slow enough to allow for intensive processing and calculation, we chose 65 MHz for the pixel clock. The full set of data is shown in Figure 9.

<table>
<thead>
<tr>
<th>Format</th>
<th>Pixel Clock (MHz)</th>
<th>Horizontal (in Pixels)</th>
<th>Vertical (in Lines)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Active Video</td>
<td>Front Porch</td>
</tr>
<tr>
<td>640x480, 60Hz</td>
<td>25.175</td>
<td>640</td>
<td>16</td>
</tr>
<tr>
<td>840x480, 72Hz</td>
<td>31.500</td>
<td>040</td>
<td>24</td>
</tr>
<tr>
<td>640x480, 75Hz</td>
<td>31.500</td>
<td>640</td>
<td>16</td>
</tr>
<tr>
<td>640x480, 85Hz</td>
<td>36.000</td>
<td>640</td>
<td>32</td>
</tr>
<tr>
<td>800x600, 60Hz</td>
<td>38.100</td>
<td>800</td>
<td>32</td>
</tr>
<tr>
<td>800x600, 60Hz</td>
<td>40.000</td>
<td>800</td>
<td>40</td>
</tr>
<tr>
<td>800x600, 72Hz</td>
<td>50.000</td>
<td>800</td>
<td>56</td>
</tr>
<tr>
<td>800x600, 75Hz</td>
<td>49.500</td>
<td>800</td>
<td>16</td>
</tr>
<tr>
<td>800x600, 85Hz</td>
<td>56.250</td>
<td>800</td>
<td>32</td>
</tr>
<tr>
<td>1024x768, 60Hz</td>
<td>65.000</td>
<td>1024</td>
<td>24</td>
</tr>
<tr>
<td>1024x768, 70Hz</td>
<td>75.000</td>
<td>1024</td>
<td>24</td>
</tr>
<tr>
<td>1024x768, 75Hz</td>
<td>78.750</td>
<td>1024</td>
<td>16</td>
</tr>
<tr>
<td>1024x768, 65Hz</td>
<td>94.500</td>
<td>1024</td>
<td>48</td>
</tr>
</tbody>
</table>

Figure 9: We could have chosen from any of the above clock setups, but settled on the 1024x768, 60Hz format. This was decided primarily because the our computer screen resolution was set to 1024x768 and we wanted to have enough clock time to process moderately intense calculations. (Source: Rick Ballantyne, Xilinx Inc.)

As suggested by its name, the vga_with_ram interfaces with the block ram memory. Under normal operation (without motion detection), the module writes to memory whenever ntsc_to_ram sends data with vwe (video write enabled) high. In addition, vga_with_ram reads from video memory at the location corresponding to current drawing position on the screen. Every memory pixel is displayed four times on a row and column, generating a 4x4 pixel block with the same luma value on the screen.
Although the image quality is degraded, it is more than sufficient for motion detection, as will be discussed later.

To perform motion detection, which uses the same memory for reading and needs sufficient clock cycles for processing, luma data storage is interrupted. When the signal “recording” is high, normal storage processes prevail. However, at the moment recording is low, motion processing starts. Memory writing is turned off and access is given to the video_processing methods.

Motion tracking begins with calculating the differences in luma values in the two time-separated frames. The algorithm is interested in the sum of absolute difference (SAD) for each 8x8 pixel macroblock. An example of SAD calculation is shown in Figure 10. A full frame has 32x12 macroblocks, and for each macroblock, an SAD value is calculated and sent to the Calc_MBError module for further processing.

![Figure 10: In this example, 4x4 pixel macroblocks outlined in red are shown with the pixel luminance in gray and white. Motion moves up (the gray area moved up 2 pixels). The motion detection algorithm measures differences in SAD values in the two frames. For each macroblock that “sees” the two-pixel up movement, SAD values will be high because the difference in luma value in the pixel positions is high.](image-url)
Section 7.b.iv Calc_MBErrow (Ray):

This module receives luma difference data from the vga_with_ram module and calculates SAD values for each of the 384 (32 * 12) macroblocks of the frame, actively keeping track of the four highest SAD blocks. When vga_with_ram finishes sending luma differences for the two frames, “done_retrieving_data” is made high, and the calc_mbeerror module computes the sum of the SADs from the four highest blocks. If the sum is past a threshold (set through testing at 25000), then signal “center_ready” is set to high and the coordinates of the four high SAD macroblocks is averaged and sent to vga_with_mem for angle calculation.
Section 7.b.v Calc_Camera_Angle (Ray):

If motion is detected, the camera needs to turn to the motion such that the target area will be centered by the camera. This module takes the 9-bit motion_center coordinate and determines deviation from the camera’s “center macroblock” (row=6, col=16). Assuming the object is around 4 feet away, it calculates using rough cases how much to turn the camera both in tilt and swivel movements. Figure 11 shows the calculation procedure.

Swivel Movement Calculation

| mbrow < 8, move 20° left | mbrow < 15, move 10° left | 15≤mbrow≤8, no movement | mbrow > 17, move 10° right | mbrow > 24, move 20° right |

Tilt Movement Calculation

| mbcoll < 4, move 10° up |
| 4 ≤ mbcoll ≤ 4, no movement |
| mbcoll > 8, move 10° down |

Figure 11: Calculation for angles after motion is detected. The Calc_Camera_Angle module takes in the motion center macroblocks coordinates and determines how much to move the camera so as to center the camera on the moving target. Calculations are performed assuming the target is 4 feet away.
Section 7.b.vi VideoRam (Ray):

The memory used in this project was a 256x192 dual-port block ram with 8 bits for luminance for each address. Pixel luminance values were stored and partitioned in two different areas on the ram, corresponding to each of the two different frames being stored. Although two separate rams could have been created (for dual access reading), a simplification was made such that displaying debug video on the screen was easier reading one ram and displaying both frames (see Figure 8).

Write enabled for the ram is controlled by vga_with_ram. Writing is disabled, or reading is enabled, when luma difference values need to be calculated. Otherwise, the write is switched off and on depending on data sent from the camera. When generating pixels for the monitor, writing is turned off.
Section 7.b.vii Testing and Debugging of the Video Module (Ray):

Debugging was mostly done through the logic analyzer, oscilloscope, and led/hex display outputs. For the state machine used to calculate recording, storeInA, and storeInB in the ntsc_to_ram module, the logic analyzer was sufficient to determine whether or not the module was working. Every module in the video processing unit has 64 bits of debug output that can be sent to the hex-display or any of the user/analyzer outputs for the logic analyzer.

Initially, the video quality was poor with much noise, which negatively affected motion detection. We corrected with several fixes. Originally, in module ntsc_decode, recording was triggered off the vsync (fvh[1]) from the camera, however the signal fvh[2] was much better defined under the oscilloscope, and when we triggered off fvh[2], results improved. Furthermore, an adjustment in the pixel clock speed to a lower clock (from 78.5MHz to 65MHz) allowed for more processing time and eliminated much of the noise in the video.

If the video processing module were designed again, we would have used ZBT memory storage instead of block memory. Not only is ZBT memory faster, but there’s more of it (4 megabytes), enough to store more than two high quality (1024x768 pixel) frames. With such memory, a more complicated algorithm could be made. For example, instead of comparing across just two frames, multi-frames could be used and we could estimate not only where the motion is, but where it is going. In addition, by using edge detection on high quality images, we can possibly make the camera zoom in on the target.

Algorithm design is very flexible, and in our case, while our motion detection worked well, there were definite limitations. We can define the general area of motion, but because every moving block can potentially create two blocks with high SAD values (one block in the location where the object is moving to and the other where the object was), exact motion detection is not particularly precise. A higher resolution detection process, but one with more processing requirements, would be to actively find motion vectors for each macroblock, and choosing an area with high vector amplitude. These vectors represent macroblock movement from one frame to the other.
Section 7.c.i Audio Processor (Bo)

The audio processor takes inputs from four amplified microphone outputs, digitizes the data from the ADC (Analog to Digital Converter), which is driven by the ADC Controller Module. The data is sent through signal processing modules (Differencer, Slopefinder, and Angle Calculator) to calculate the direction of a sound source. This angle is output to the Master Controller, which drives the underlying motor to point the camera at the sound source. This angle is output along with other intermediate angles from Slopefinder instances for debugging purposes, as shown in the block diagram Figure 12. The timer modules are used to accurately drive the ADC, which has stringent timing constraints.

Figure 12: Block diagram of the Audio Processing Module. Analog mic inputs are converted to discrete data, which is used to calculate the location of the sound source by signal processing modules.
Section 7.c.ii Microphones, Amplifiers and ADCs (Bo)

The physical components of the Audio Processing module are the microphones, microphone amplifiers and Analog to Digital Converter chips. The electret microphones have an output range of only a few millivolts, so they need to be amplified for usable signals to be detected by the ADC. The 30dB signal amplifier converts the microphone outputs to an approximately 0-5V range signal.

![Figure 13: The microphone amplifier and AD7871 ADC setup, which is implemented four times (one for each mic). The Microphone amplifier output is fed into the ADC. The data bits are output to the FPGA labkit, and timing signals from the labkit are input to the ADC.](image)

As shown in Figure 13, the microphone amplifier signal is input to the V_in pin of the ADC. The ADCs are Analog Devices AD7871 chips, which output 14-bit data at a maximum of 83K samples per second. The data is output to the FPGA labkit through userpins for processing. The ADC controller drives the ADC through three timing signals (RD_BAR, CS_BAR, and CONVST_BAR), which will be detailed below.

The clock pin of the ADC is tied to V_{ss} (-5V) in order to select the internal laser-trimmed 2MHz clock, which the ADC will use to time its analog to digital conversion cycles. The 14/8_bar/clock pin of the ADC is tied to V_{dd} (+5V), which selects the mode for parallel output. The alternative is to output the data serially. We selected parallel mode because there were less timing issues; once the conversion is complete, all data bits are available for reading.

Pin NC is a No-Connect pin. C_{ref} is the decoupling point for on-chip reference. It is permanently tied to ground with a 10nF capacitor. Ref_{out} is a 3V output.
Section 7.c.iii Analog to Digital Converter Controller (Bo)

The ADC Controller is responsible for transmitting and receiving signals from the ADC chips. In order for the ADC to operate correctly, the signals RD_BAR, CS_BAR, and CONVST_BAR need to be timed properly.

Figure 14: Timing Diagram for AD7871 Analog to Digital Converter. Low CONVST_BAR begins conversion, low INT_BAR signals end of conversion, and low RD_BAR and CS_BAR generate valid data.

Figure 14 shows the timing diagram for the AD7871. When CONVST_BAR is pulsed low, the ADC begins its conversion on the rising edge of the pulse. CONVST_BAR must be low for at least 50ns in order for the ADC to recognize it as a valid low pulse, as indicated in Table 1, the timing characteristics table. After the conversion is complete, the ADC switches INT_BAR low, and waits for the rising edge of a synchronous low RD_BAR and CS_BAR pulse before switching INT_BAR high again. RD_BAR and CS_BAR must be held low for at least 60 ns in order for valid data to be generated. 57 ns after the falling edge of the RD_BAR and CS_BAR pulses, the 14-bit data becomes valid, and will stay valid until 5ns after the rising edges of RD_BAR and CS_BAR.

Table 4: Timing Characteristics of AD7871 Analog to Digital Converter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Limit at T_{MIN}, T_{MAX} (J, K, A, B Versions)</th>
<th>Limit at T_{MIN}, T_{MAX} (T Version)</th>
<th>Units</th>
<th>Conditions/Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_1</td>
<td>50</td>
<td>50</td>
<td>ns min</td>
<td>CONVST Pulse Width</td>
</tr>
<tr>
<td>t_2</td>
<td>0</td>
<td>0</td>
<td>ns min</td>
<td>CS to RD Setup Time (Mode 1)</td>
</tr>
<tr>
<td>t_3</td>
<td>60</td>
<td>75</td>
<td>ns min</td>
<td>RD Pulse Width</td>
</tr>
<tr>
<td>t_4</td>
<td>0</td>
<td>0</td>
<td>ns min</td>
<td>CS to RD Hold Time (Mode 1)</td>
</tr>
<tr>
<td>t_5</td>
<td>70</td>
<td>70</td>
<td>ns min</td>
<td>RD to INT Delay</td>
</tr>
<tr>
<td>t_6</td>
<td>57</td>
<td>70</td>
<td>ns max</td>
<td>Data Access Time after RD</td>
</tr>
<tr>
<td>t_7</td>
<td>5</td>
<td>5</td>
<td>ns min</td>
<td>Bus Relinquish Time after RD</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>50</td>
<td>ns max</td>
<td></td>
</tr>
</tbody>
</table>
Each sample conversion begins with the low CONVST_BAR pulse. Therefore, in order to have consistently spaced samples, CONVST_BAR must be clocked. Timer1 counts for 960 65MHz clock cycles, or 15 µs, and then pulses the expired1 signal high for one clock cycle. It is at this high pulse that CONVST_BAR starts to be pulsed low.

To ensure that output signals obey the timing characteristics detailed above, the amount of time that each signal is pulsed must be controlled accurately. The output pulses have no upper bound restrictions; only lower bound. Therefore, a universal pulse timer that counts for at least the highest lower bound fulfills the role. Timer2 counts for 10 65MHz clock cycles, or 154 ns, which is well above the highest lower bound of 60 ns. In implementation, the low pulses of CONVST_BAR, CS_BAR, and RD_BAR are timed with Timer2, and get switched back to their high state after expired2 of Timer2 pulses high.

Because there are four ADCs that need to be driven, a synchronous system to drive all four at once is the most sensible option. If all four ADCs are asynchronous, data would be valid at different times, throwing off the synchronous system that the rest of the modules rely on to operate. Unfortunately, the amount of time to compute one sample is not fixed; it depends on the amplitude of the analog signal. Therefore, INT_BAR (which signals the end of conversion) arrives at different times for each ADC for each sample.

In order to drive the ADCs synchronously, the RD_BAR and CS_BAR pulse signals cannot depend on INT_BAR. They must be pulsed after a consistent time period after conversion starts, as shown in the Logic Analyzer output in Figure 15. This time period must be a safe amount that is longer than the maximum time conversion can take. The spec sheet for the AD7871 indicates 12 µs as the maximum conversion time; in our project, Timer3, which clocks this simulated time period, switches expired3 to high after 14 µs.

With this scheme, the maximum sampling rate cannot be achieved, because maximum sampling rate is only possible if RD_BAR and CS_BAR are pulsed almost immediately after INT_BAR falls low. However, this scheme does successfully synchronously drive the four ADCs, which is a much more important feature to have. Even with a lower sampling rate, each sample arrives every 15 µs. The speed of sound in air is 1180 ft/s,
equating to one inch every 70µs. Therefore, the resolution of our 67K samples/second system is approximately 0.2 in, which is very reasonably accurate for our purposes.

In terms of actual implementation, the ADC Controller is a Finite State Machine. It has five states, all of which are sequenced forward by timers. There is only one path for the states to progress along, as shown in Figure 16.

Figure 16: Finite State Machine of the ADC Controller. The states represent different time periods of the ADC timing diagram. The signals are output to all four ADCs at once to satisfy the synchronous system. (“SIGNAL{1,2,3,4} = X” indicates value X is applied to SIGNAL1, SIGNAL 2, SIGNAL 3, and SIGNAL4)

State 0: (Begin A to D conversion)

The ADC controller is beginning the conversion process. It starts sets CONVST_BAR low, while keeping CS_BAR and RD_BAR high. The system progresses to State 1 when expired2 from Timer2 reaches high, at 10 65MHz clock cycles after entering State 0.

State 1: (A to D conversion in process)

The ADC controller ends the low pulse applied to CONVST_BAR. It resets CONVST_BAR high, and maintains CS_BAR and RD_BAR high. The system progresses to State 2 when expired3 from Timer3 reaches high, at 912 65MHz clock cycles after entering State 1.
State 2: (A to D conversion finished)

The ADC controller prepares for valid data. It keeps CONVST_BAR high, and sets CS_BAR and RD_BAR low. The system progresses to State 3 when expired2 from Timer2 reaches high, at 10 65MHz clock cycles after entering State 2.

State 3: (Valid data ready for reading)

The ADC controller is in valid data mode. It keeps CONVST_BAR high, and keeps CS_BAR and RD_BAR low. The system progresses to State 4 when expired2 from Timer2 reaches high, at 10 65MHz clock cycles after entering State 3.

State 4: (Wait for next conversion cycle)

The ADC controller is in waiting mode. It keeps CONVST_BAR high, and flips CS_BAR and RD_BAR high. The system progresses back to State 0 when expired1 from Timer1 reaches high, at 960 65MHz clock cycles after entering State 0 four states ago.
Section 7.c.iv Sound Source Location Algorithm (Bo)

For purposes of clarity and comprehension, the algorithm for sound source location we have developed will be explained here.

When a loud, sharp noise is produced near the sentry machine, different microphones pick up the sound at different times because they are located at various distances from the sound source. The values of these time differences carry important information about the relative location of the sound source to the microphones. In the engineering field, this method of localizing sound is termed “Time Difference of Arrival” or TDOA.

With two microphones at a known and fixed distance apart, we can approximately constrain the sound source to be from two linear vectors. The derivation begins below.

![Diagram of TDOA](image)

**Figure 17**: Triangle model of TDOA (Time Difference of Arrival). The capital letters A, B, and C represent the angles of the triangle, and the lower case letters a, b, and c represent the distance/time differences between the vertices.

As shown in Figure 17, the sound source is closer to A than to B. Thus, the time it takes for sound to travel from the source to A is also less than the time it takes to travel to B. In fact, because speed = distance / time, and speed is constant, distance and time are proportional. We can therefore conceptualize the distances a, b and c to also represent the time it would take for sound to travel between the vertices.

Let $a = b + d$, where $d$ is the TDOA between point B and point A, the two microphones.

Using the Law of Cosines, we get $A = \cos^{-1}\left(\frac{(b + d)^2 - b^2 - c^2}{-2bc}\right)$
Superimposing this figure on a Cartesian coordinate system, with vertex A at the origin (0,0), we get the location of the sound source to be at \((x,y) = (b\cos(A), b\sin(A))\), which results in:

\[
(x,y) = \left( \frac{(b + d)^2 - b^2 - c^2}{-2c}, \pm b \sqrt{1 - \left( \frac{(b + d)^2 - b^2 - c^2}{2bc} \right)^2} \right)
\]

To linearize the model for faster computation, we must calculate the slope of the lines. As the distance \(b\) becomes larger than the values \(c\) and \(d\), the slope \(m\) of the lines approach:

\[
m = \sqrt{1 - \left( \frac{d}{c} \right)^2}
\]

The initial values of \(x\) and \(y\) are linearized to be \(x_0 = \frac{c - d}{2}\) and \(y_0 = 0\).

Thus the final linearized model becomes:

\[
(x,y) = \left\{ \frac{c - d}{2}, \pm b \sqrt{1 - \left( \frac{d}{c} \right)^2} \right\}, \text{ for all } b \geq 0.
\]

Figure 18 depicts one possible solution for the two-microphone linearized model. The sound source for a particular \(c\) and a particular \(d\) lies along the two vectors.
Figure 18: The possible locations for the sound source exist on the two vectors, for a measured \( d \) delay and predetermined \( c \) distance between the microphones.

With two microphones, two possible paths are calculated. By adding two more microphones, and positioning them correctly, we can identify the exact location of the sound source in a two-dimensional plane.

The same calculations are applied on the second pair of microphone data, except translating the figure onto the Cartesian coordinate system requires switching the \( x \) and \( y \) axes, reflecting the new \( y \) axis, and adding \( c/2 \) to both \( x \) and \( y \) coordinates:

\[
(x_2, y_2) = \left\{ \frac{c}{2} \pm \frac{b}{c} \sqrt{1 - \left( \frac{d_2}{c} \right)^2}, \frac{c}{2} + \frac{d_2 - c}{2} \pm \frac{b}{c} \right\}, \text{ for all real } b \geq 0.
\]

The intersection point of the two pairs of vectors can be calculated through solving the two simultaneous solutions for equivalent \( x \) and \( y \) coordinates. The idealized model for such a solution is depicted in Figure 19.
Figure 19: The solutions of the individual pairs of microphones superimposed on each other result in an intersection point, which represents the location of the sound source.
Section 7.c.v Differencer (Bo)

The implementation of the algorithm described above is split up into multiple modules, each with its mathematical contribution. The Differencer module continuously takes pairs of 14-bit data and calculates the TDOA between the two microphones when the sound is valid. A noise is valid if its amplitude is above a certain threshold. In our specific implementation, that threshold is $0h1554$, or correspondingly, $2V$ from the amplifier output. Two Differencer modules are instantiated for our project, one for each opposing pair of microphones.

The Differencer module takes is implemented as a Finite State Machine, with four states:

**State 0: (Wait for valid sound)**

No registers change value in this state. If the module was just reset, then $\text{dir12}$, $\text{delay}$, and $\text{count}$ all stay at the value 0. $\text{dir12}$ indicates the general direction of sound; it is 1 if $\text{db1}$ (the first set of data) passes the threshold first, and 2 if $\text{db2}$ passes the threshold first. $\text{delay}$, when nonzero, represents the TDOA between the two microphones for a valid sound. $\text{count}$ is an internal counter register which times the TDOA, and stores its value into $\text{delay}$ when the second microphone data set becomes valid.

The state changes to State 1 if it detects $\text{db1}$ passing the threshold, and changes to State 2 if $\text{db2}$ passes the threshold. During the same clock cycle of state change, $\text{dir12}$ changes as described above, and $\text{count}$ is reset to 0.

**State 1: (db1 valid sound)**

In State 1, Differencer is waiting for $\text{db2}$ to pass the threshold and become valid. During the wait, it increments $\text{count}$ register and maintains $\text{dir12}$ at 1. If $\text{db2}$ does not become valid before 15ms (1023 samples) after entrance to State 1, then the sound itself is not valid, and the state reverts to State 0. $\text{dir12}$ also reverts to 0.

If $\text{db2}$ does pass the threshold, the current value of $\text{count}$ is stored to $\text{delay}$, and the state changes to State 3.

**State 2: (db2 valid sound)**

In State 2, Differencer is waiting for $\text{db1}$ to pass the threshold and become valid. During the wait, it increments $\text{count}$ register and maintains $\text{dir12}$ at 2. If $\text{db1}$ does not become valid before 15ms (1023 samples) after entrance to State 2, then the sound itself is not valid, and the state reverts to State 0. $\text{dir12}$ also reverts to 0.
If \( db1 \) does pass the threshold, the current value of count is stored to delay, and the state changes to State 3.

**State 3: (valid delay, wait for reset)**

In State 3, all registers are static at the value they were when State 3 was entered. Because a valid delay was computed, the motor during this state is moving the sentry machine to the angle computed in modules further down the pipeline. The Master Controller module cannot accept any new angles resulting from new delay values until the motor has arrived at its intended angle and has stopped moving. The Master Controller sends a reset signal to the entire Audio Processing module when this occurs, to receive a new angle.
Section 7.c.vi Slopefinder (Bo)

The Slopefinder module takes the delay and dir12 values calculated by the Differencer module for one pair of microphones, and outputs the possible vector angles that the sound could be from, as well as the initial \( x \) value.

Slopefinder uses simple arithmetic to calculate \( x_0 = \frac{c - d}{2} \). In our implementation, \( c \) is fixed to be 120 and \( d \) is the delay input. For the angles, it uses a manually-entered lookup table to map the computed \( \frac{d}{c} \) value to find an approximate slope, represented by \[ \sqrt{1 - \left( \frac{d}{c} \right)^2} \]. The \( \frac{d}{c} \) value is calculated by a Coregen divider module. In order to accurately calculate this value to the hundredth decimal place, the dividend \( d \) was multiplied by 10, and \( c \) was replaced by 12 (effectively divided by 10). The resulting quotient was 100 times the size of the actual value \( \frac{d}{c} \), but because the new quotient is an integer, it can be used easily in the lookup table with great accuracy.

angle_out1 and angle_out2 represent the two possible vector angles, and are output to the Master Controller as debug display angles in the format of a 6-bit number, and range from 0 to 35. This format is specified by the Master Controller. Each of these numbers represents 10 degrees. posangle is output to the Angle Calculator module, and only represents the positive slope in 10-degree increments. Like the Differencer modules, two Slopefinder modules are instantiated, each responsible for one opposing pair of microphone outputs.
Section 7.c.vii Angle Calculator (Bo)

The Angle Calculator module computes an approximate final angle for the motor to rotate to. It takes the posangle outputs from Slopefinder1 and Slopefinder2 instances and averages them to form a reasonable positive angle of the sound source. Angle Calculator also uses the dir12 outputs from Differencer to locate the correct quadrant the sound is coming from. With these direction and average angle values, a final angle is selected and is output to the Master Controller module.
Section 7.c.viii Testing and Debugging the Audio (Bo)

In order to effectively test and debug a digital system, one must be confident that the error is actually in the digital realm, and not a result of flaky physical components. At the beginning of the project, lots of messy wires and semi-loose connectors were linking the analog components to the digital labkit, causing many problems even when the digital system was working properly. After upgrading to ribbon cable, alternating ground wires, and solid header connectors, many problems went away and we could focus on the digital system design.

Digital systems that rely heavily on real-world analog signals need to have any assumed real-world parameters and other constants validated with testing. The ringing amplitude of a hand clap is not a perfect impulse, so treating it like one in the digital system is a critical mistake. We performed extensive tests with the oscilloscope to discover a valid sound threshold at which TDOAs were consistent.

Along the entire project, a common practice we adhered to was to display as much debugging information as possible. If one value appears to be incorrect, the root of the problem cannot be determined without feedback on other values that the incorrect component depended on. One extremely important debugging tool is the video debug screen shown in a picture in Section 7.a.iv. The upper-left quadrant (when valid data is input, not the case in picture) shows the two pairs of vectors on which the sound source could be located. This tool displays a large amount of information in an efficient way, and although it took a while to create, the debugging time it saved made the tool worth it.
Section 8 Conclusion

We successfully implemented most of the features we originally wanted in the sentry security machine. Both audio and visual components could detect a target and send the signals to the motor control to focus in on the object. The integration of all three components, along with the various debug display modules, was demonstrated to the TA’s and instructors on several different occasions. However, we did also experience a myriad of problems while designing and building the machine. The finished product experienced some minor failures as well.

Our reliance on relatively cheap (and common in the lab), but malfunction prone, stepper motors was a mechanical issue that we did not anticipate. The weight of the camera, mounting board, and four proto-boards were too much to handle for the stepper motor and both our motors eventually malfunctioned. In addition, the tilt stepper motor never had enough stability to hold the camera in quasi-equilibrium, so we had to abandon vertical motion altogether. The mechanical obstacles of allowing 360 degree motion without entangling wires also impeded our original hopes of fully automatic motion.

Furthermore, in the video processing module, zero-bus turnaround (ZBT memory should have been used instead of Block RAM. Although the algorithm tracked motion well, against similar color background and foreground, movement did not increase past the noise-rejecting threshold. A higher resolution image can fix that (threshold can be lowered without increasing the probability of false alarm), but only with a higher capacity memory such as the ZBT.

The Audio Processor successfully computed a reasonable angle to locate the sound source. The modules worked well not only within the Audio Processor but also with other modules via the Master Controller. However, the ideal exact position locator (commented out “newlocator” module) did not calculate correct x and y coordinates, which was a result of not having enough time to debug the complex arithmetic operations. The process of creating the Audio Processor taught us a great deal of what was necessary to create a robust digital system. Dealing with problems such as driving four separate ADC chips forced us to think of creative synchronous solutions. Optimization of algorithms to produce low-latency results, such as linearizing models and using lookup tables instead of real-time computation was an important lesson learned.

Debugging took the longest part of the whole project. While there were certainly tools (logic analyzers, oscilloscopes, TA’s) that were available for help, much of the debugging was made easier on keeping the design and code simple. On several occasions we rewrote entire modules of code from scratch, using completely different algorithms to fix the problem. Thankfully, on most occasions our errors were caused by small mistakes that could be caught with patience and careful observation of the debugging data.

We expected the project to be challenging, but were well rewarded with the result. Our project taught us much about sound design, “keeping it simple”, and debugging methods and tools. This project is the culmination of one semester of learning verilog programming and digital design, and we are very happy with the completion of the sentry security machine. On a last note, we would like to thank the countless hours the TA’s (Jae, Javier, Willie, and Eric), lab instructor (Gim), and lecturers (Prof. Chuang and Prof. Terman) spent helping us design and debug the project.
Section 9 Appendix: Verilog Code

/////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
/*
6.111 Final Project
Created by Robert Speaker, Ray Wu, and Bo Zhu
This is the master controller that integrates all the subcomponents of the sentry security system with the appropriate labkit connections.

User inputs:
Switches
Number
0 Manual Override (Active High)
1 Debug Angle Mode (Active High)
2 Unused
3 Continuous Video Feed (Active High)
4 High Contrast (Active High)
5 Audio/Video Priority (Audio: High, Video: Low)
6 Video Processing display (Active High)
7 Picture Viewer/Debug module (Picture: High, Debug: Low)

Buttons
Enter: User Reset
0: Take Picture
1: Unused
2: Simulate Audio Motion
3: Simulate Video Motion
Up: Move camera up if manual override is asserted
Down: Move camera down if manual override is asserted
Left: Move camera left if manual override is asserted
Right: Move camera right if manual override is asserted
*/

module master_controller (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock,
                         vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
                         vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
                         vga_out_vsync,
                         tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
                         tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
                         tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,
                         tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
                         tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
                         tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
                         tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,
                         ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
                         ram0_ce_b, ram0_we_b, ram0_bwe_B,
                         rami_data, rami_address, rami_adv_ld, rami_clk, rami_cen_b,
                         rami_ce_b, rami_ce_b, rami_we_b, rami_bwe_B,
                         clock_feedback_out, clock_feedback_in,
                         flash_data, flash_address, flash_ce_b, flash_we_b, flash_we_b,
flash_reset_b, flash_sts, flash_byte_b,
rs232_txd, rs232_rxd, rs232_rts, rs232_cts,
mouse_clock, mouse_data, keyboard_clock, keyboard_data,
clock_27mhz, clock1, clock2,
disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
disp_reset_b, disp_data_in,
button0, button1, button2, button3, button_enter, button_right,
button_left, button_down, button_up,
switch,
led,
user1, user2, user3, user4,
daughtercard,
	systemace_data, systemace_address, systemace_ce_b,
	systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

 analyzer1_data, analyzer1_clock,
 analyzer2_data, analyzer2_clock,
 analyzer3_data, analyzer3_clock,
 analyzer4_data, analyzer4_clock);
input  clock_27mhz, clock1, clock2;

output  disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
input  disp_data_in;
output  disp_data_out;

input  button0, button1, button2, button3, button_enter, button_right,
        button_left, button_down, button_up;
input  [7:0] switch;
output  [7:0] led;

input  [31:0] user1, user2, user4; //use 1, 2, and 4 as input, 3 for output
output [31:0] user3;

inout [43:0] daughtercard;
inout [15:0] systemace_data;
output [6:0] systemace_address;
output systemace_ce_b, systemace_we_b, systemace_oe_b;
inout systemace_irq, systemace_mpbrdy;
output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
              analyzer4_data;
output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

////////////////////////////////////////////////////////////////////////////////
// I/O Assignments
////////////////////////////////////////////////////////////////////////////////

// Audio Input and Output
assign beep= 1'b0;
assign audio_reset_b = 1'b0;
assign ac97_synch = 1'b0;
assign ac97_sdata_out = 1'b0;
// ac97_sdata_in is an input

// Video Output
assign tv_out_ycrcb = 10'h0;
assign tv_out_reset_b = 1'b0;
assign tv_out_clock = 1'b0;
assign tv_out_i2c_clock = 1'b0;
assign tv_out_i2c_data = 1'b0;
assign tv_out_pal_ntsc = 1'b0;
assign tv_out_hsync_b = 1'b1;
assign tv_out_vsync_b = 1'b1;
assign tv_out_blank_b = 1'b1;
assign tv_out_subcar_reset = 1'b0;

// Video Input
assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b1;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b1;
//assign tv_in_reset_b = 1'b0;
assign tv_in_clock = clock_27mhz; //1'b0;
//assign tv_in_i2c_data = 1'bZ;
// tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

// SRAMs
// enable ram0 bank to store pictures in ZBT memory
*assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_clk = 1'b0;
assign ram0_we_b = 1'b0;
assign ram0_cen_b = 1'b1;*/
assign ram0_adv_ld = 1'b0;
assign ram0_ce_b = 1'b0;
assign ram0_oe_b = 1'b0;
assign ram0_bwe_b = 4'h0;

assign rami_data = 36'hZ;
assign rami_address = 19'h0;
assign rami_adv_ld = 1'b0;
assign rami_clk = 1'b0;
assign rami_cen_b = 1'b1;
assign rami_ce_b = 1'b1;
assign rami_oe_b = 1'b1;
assign rami_we_b = 1'b1;
assign rami_bwe_b = 4'hF;

assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input

// Flash ROM
assign flash_data = 16'hZ;
assign flash_address = 24'h0;
assign flash_ce_b = 1'b1;
assign flash_oe_b = 1'b1;
assign flash_we_b = 1'b0;
assign flash_reset_b = 1'b0;
assign flash_byte_b = 1'b1;
// flash_sts is an input

// RS-232 Interface
assign rs232_txd = 1'b1;
assign rs232_rts = 1'b1;
// rs232_rxd and rs232_cts are inputs

// PS/2 Ports
// mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

// LED Displays
/*@ assign disp_blank = 1'b1;
assign disp_clock = 1'b0;
assign disp_rs = 1'b0;
assign disp_ce_b = 1'b1;
assign disp_reset_b = 1'b0;
assign disp_data_out = 1'b0;*/
// disp_data_in is an input

// Buttons, Switches, and Individual LEDs
// assign led = 8'hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs

// User I/Os
//assign user1 = 32'hZ;
//assign user2 = 32'hZ;
//assign user3 = 32'hZ;
//assign user4 = 32'hZ;

// Daughtercard Connectors
assign daughtercard = 44'hZ;

// SystemACE Microprocessor Port
assign systemace_data = 16'hZ;
assign systemace_address = 7'h0;
assign systemace_ce_b = 1'b1;
assign systemace_we_b = 1'b1;
assign systemace_oe_b = 1'b1;
// systemace_irq and systemace_mpbrdy are inputs

// Ray uses logic analyzer signals:
// Logic Analyzer
//assign analyzer1_data = 16'h0;
//assign analyzer1_clock = 1'b1;
Speaker, Wu, Zhu

```verilog
assign analyzer1_clock = clock_27mhz;
//assign analyzer2_data = 16'h0000;
assign analyzer2_clock = 1'b1;
assign analyzer3_clock = tv_in_line_clock1;
assign analyzer3_data = 16'h0;
assign analyzer4_data = 16'h0;
assign analyzer4_clock = 1'b1;

// use FPGA's digital clock manager to produce a 65MHz clock (actually 64.8MHz)
wire clock_65mhz_unbuf,clock_65mhz;
DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
// synthesis attribute CLKFX_DIVIDE of vclk1 is 10
// synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37
BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

// power-on reset generation
wire power_on_reset; // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
,A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

// ENTER button is user reset
wire reset,user_reset;
debounce deb1(power_on_reset, clock_65mhz, ~button_enter, user_reset);
assign reset = user_reset | power_on_reset;

//Ray's code:
wire [15:0] vram_addr;
wire [7:0] vram_data_in;
wire [7:0] vram_data_out;
wire vram_clk;
wire vram_we;

wire recording; //high when skipping a frame
wire center_ready;
wire [3:0] center_mb_row;
wire [4:0] center_mb_col;
wire [12:0] debug; //used for debug
wire [8:0] mb_output;
wire [63:0] debug_vga; //ray wu debug(display_pix_per_line)
wire [7:0] rays_red, rays_green, rays_blue;
wire rays_hs, rays_vs, rays_b, rays_pixel_clock, rays_vga_out_sync_b;
wire [5:0] rays_v_angle_v, rays_v_angle_h, bos_a_angle, bos_a_angle1, bos_a_angle2, bos_a_angle3, bos_a_angle4;
wire rays_v_motion, bos_a_motion;
vga_with_ram vr(.reset(reset), .clock_27mhz(clock_27mhz), .clock_65mhz(clock_65mhz),
, .vga_out_red(rays_red), .vga_out_green(rays_green),
, .vga_out_blue(rays_blue),
, .vga_out_sync_b(rays_vga_out_sync_b), .vga_out_blank_b(rays_b),
, .vga_out_pixel_clock(rays_pixel_clock),
, .vga_out_hsync(rays_hs), .vga_out_vsync(rays_vs),
, .vram_addr(vram_addr), .vram_data_in(vram_data_in),
, .vram_data_out(vram_data_out),
, .vram_clk(vram_clk), .vram_we(vram_we), .recording(recording),
, .center_ready(center_ready), .motion_center(center_mb_row, center_mb_col),
, .v_angle_h(rays_v_angle_h), .v_angle_v(rays_v_angle_v),
, .angle_calculated(rays_v_motion),
, .debug(debug_vga), .mb_output(mb_output));

// ADV7185 NTSC decoder interface code
// adv7185 initialization module
adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz),
    .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
    .tv_in_i2c_clock(tv_in_i2c_clock),
    .tv_in_i2c_data(tv_in_i2c_data));

wire [29:0] ycrcb; // video data (luminance, chrominance)
wire [2:0] fvh; // sync for field, vertical, horizontal
wire dv; // data valid

ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
    .tv_in_ycrcb(tv_in_ycrcb[19:10]),
    .ycrcb(ycrcb), .f(fvh[2]),
    .v(fvh[1]), .h(fvh[0]), .data_valid(dv));

// display memory: test pattern or NTSC video
//assign led = ~vram_data_out;

wire [15:0] vaddr3;
wire [7:0] vdata3;
wire vclk3;
wire vwe3;
vid_test_pat vp3 (clock_27mhz,vaddr3,vdata3,vclk3,vwe3);

wire [15:0] vaddr4;
wire [7:0] vdata4;
wire vwe4;
//wire vclk4;
wire [63:0] debug_ntsc_to_ram;
wire [5:0] debug_led6_ntsc_to_ram;
wire [8:0] logic_output;
ntsc_to_ram vp4 (.reset(reset), .clk(tv_in_line_clock1),
    .fvh(fvh), .dv(dv), .din(ycrcb[29:22]),
    .vaddr(vaddr4), .vwe(vwe4), .vdata(vdata4),
    .high_contrast(switch[4]), .display_one_frame(switch[3]),
    .debug_output(debug_ntsc_to_ram),
    .led_output6(debug_led6_ntsc_to_ram), .logic_output(logic_output),
    .skippedFrame(recording));

// select video source
//wire video_source;
//assign video_source = 0; //always use NTSC video
debounce vid_source(reset, clock_65mhz, switch[1], video_source);
assign vram_addr = video_source ? vaddr3 : vaddr4; // fill video RAM with NTSC video
data or b/w bars
assign vram_data_in = video_source ? vdata3 : vdata4;
assign vram_clk = video_source ? vclk3 : tv_in_line_clock1;
assign vram_we = video_source ? vwe3 : vwe4;/*/

assign vram_addr = vaddr4; // fill video RAM with NTSC video data
assign vram_data_in = vdata4;
assign vram_clk = tv_in_line_clock1;
assign vram_we = vwe4;

// debugging
assign analyzer2_data[15:0] = {2'b00, vwe4, fvh, tv_in_ycrcb[19:10]};

parameter VERSION = 8'd51;
wire [63:0] RaysHexData;
assign RaysHexData[63:0] = debug_vga[63:0]; // {56'b0,VERSION};
//*display_16hex my_display(reset, clock_27mhz, debug_vga, // {56'b0,VERSION},
disp_blank, disp_clock, disp_rs,
disp_ce_b, disp_reset_b, disp_data_out);*/

//assign led[7] = center_ready;
//assign users:
// assign user2[7:0] = debug_vga[7:0]; //difference_luma
//assign user2[31:8] = 24'b0;
//assign user3[31:9] = 23'b0;
//user3[8:0] come from ntsc_to_ram

//end Ray's code

// generate basic XVGA video signals
wire [10:0] hcount;
wire [9:0] vcount;
wire hsync,vsync,blank;
xvga xvga1(clock_65mhz,hcount,vcount,hsync,vsync,blank);

// feed XVGA signals to display module
wire [2:0] pixel;
wire dhsync,dvsync,dblank;

// wire up to ZBT ram
wire [35:0] z_write_data;
wire [35:0] z_read_data;
wire [18:0] z_addr;
wire z_we;

zbtt_611 zbt1(clock_65mhz, 1'b1, z_we, z_addr,
    z_write_data, z_read_data,
    ram0_clk, ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

// generate pixel value from reading ZBT memory
wire [7:0] z_pixel;
wire [18:0] z_addr1;
vram_display vd1(reset,clock_65mhz,hcount,vcount,z_pixel,z_addr1,z_read_data);

// code to write NTSC data to video memory
wire [18:0] ntsc_addr;
wire [35:0] ntsc_data;
wire ntsc_we;
ntsc_to_zbt n2z (clock_65mhz, tv_in_line_clock1, fvh, dv, ycrcb[29:22],
    ntsc_addr, ntsc_data, ntsc_we, 1'b0); //switch[3]); don't need the
debbuging switch

// code to write pattern to ZBT memory
reg [31:0] count;
always @(posedge clock_65mhz) count <= reset ? 0 : count + 1;

//wire [18:0] z_addr2 = count[0+18:0];
//wire [35:0] vpat = ( switch[1] ? {4{count[3+3:3],4'b0}} :
//{4{count[3+4:4],4'b0}} ); //switch 1 selects between test bar periods
/* wire [35:0] vpat = {4{count[3+3:3],4'b0}}; //switch 1 selects between test bar
periods

// mux selecting read/write to memory based on which write-enable is chosen
wire sw_ntsc = -switch[2]; //switches between ntsc and test bars
wire my_we = sw_ntsc ? (hcount[1:0]==2'd2) : blank;
wire [18:0] write_addr = sw_ntsc ? ntsc_addr : z_addr2;
wire [35:0] write_data = sw_ntsc ? ntsc_data : vpat; /*

wire my_we = (hcount[1:0] == 2'd2);
wire [18:0] write_addr = ntsc_addr;
wire [35:0] write_data = ntsc_data;
assign z_addr = my_we ? write_addr : z_addr1;

wire manual_override;
debounce manual(reset, clock_65mhz, switch[0], manual_override);

//generate write enable signal when user presses snapshot button
wire snapshot, picture_we; //frame_expire;
synchronize snapper(clock_65mhz, ~button0, snapshot);
//if manual override is set use button0 to take pictures
//otherwise pictures will be taken whenever visual motion is detected
reg [26:0] pic_delay_count = 0;
always @ (posedge clock_65mhz) begin
    //take a picture at startup or whenever motion is detected
    if (reset || rays_v_motion) pic_delay_count <= 26’d6500000;  //hold high for 0.1s
    else if (pic_delay_count > 0) pic_delay_count <= pic_delay_count - 1;
    else pic_delay_count <= 0;
end
assign picture_we = manual_override ? snapshot : (pic_delay_count != 0);
//assign picture_we = snapshot;
assign z_we = picture_we && my_we;
assign z_write_data = write_data;
// select output pixel data
wire Picture_b, Picture_hs, Picture_vs;
delayN dn1(clock_65mhz,hsync,Picture_hs); // delay by 3 cycles to sync with ZBT read
delayN dn2(clock_65mhz,vsync,Picture_vs);
delayN dn3(clock_65mhz,blank,Picture_b);
//reg [7:0] Picture_Pixel;
//filter out everything but the picture
parameter HMIN = 64;
parameter HMAX = 600;
parameter VMIN = 64;
parameter VMAX = 400;
wire [7:0] Picture_Pixel;
wire picture_select;
assign picture_select = hcount > HMIN && hcount < HMAX && vcount > VMIN && vcount < VMAX;
//assign Picture_Pixel[7:0] = z_pixel[7:0];
assign Picture_Pixel[7:0] = picture_select ? z_pixel[7:0] : 8’d0;
//debounce pp(clock_65mhz, reset, switch[4], picture_select);
/*always @(posedge clock_65mhz)
begin
    Picture_Pixel <= picture_select ? {hcount[8:6],5’b0} : z_pixel;
end*/
wire [5:0] v_angle_v, v_angle_h, a_angle; //angles received from video and audio processing units
wire [5:0] a_angle1, a_angle2, a_angle3, a_angle4; //all possible locations of the sound before triangulation
wire v_motion, a_motion; //used to determine whether the angles are valid
//debug angles used to demo motion:
wire [5:0] dbg_v_angle_v, dbg_v_angle_h, dbg_a_angle, dbg_a_angle2, dbg_a_angle3, dbg_a_angle4;
wire dbg_v_motion, dbg_a_motion;
wire dbg_sw; //switch to toggle between debug mode and automatic mode
debounce dbg_switch(reset, clock_65mhz, switch[1], dbg_sw);
//debug code to test automatic motion
wire v_finished_moving, h_finished_moving; //output of motion calculator when motor stops moving
wire v_reset, h_reset;
assign v_reset = reset || v_finished_moving; //reset the angles whenever the motor has already moved
assign h_reset = reset || h_finished_moving;
test_angle_generator tag1(clock_65mhz, v_reset, dbg_v_angle_v);
defparam tag1.INCREMENT = 2;
defparam tag1.START_VALUE = 18;
test_angle_generator tag2(clock_65mhz, h_reset, dbg_v_angle_h);
defparam tag2.INCREMENT = 1;
defparam tag2.START_VALUE = 33;
test_angle_generator tag2(clock_65mhz, h_reset, dbg_a_angle);
defparam tag2.INCREMENT = 2;
defparam tag3.START_VALUE = 9;
test_angle_generator tag3(clock_65mhz, h_reset, dbg_a_angle2);
defparam tag3.INCREMENT = 2;
defparam tag4.START_VALUE = 14;
test_angle_generator tag4(clock_65mhz, h_reset, dbg_a_angle3);
defparam tag4.INCREMENT = 3;
defparam tag5.START_VALUE = 0;
test_angle_generator tag5(clock_65mhz, h_reset, dbg_a_angle4);
defparam tag5.INCREMENT = 1;
defparam tag6.START_VALUE = 30;

assign v_angle_h[5:0] = dbg_sw ? dbg_v_angle_h[5:0] : rays_v_angle_h[5:0];
assign a_angle[5:0] = dbg_sw ? dbg_a_angle[5:0] : bos_a_angle[5:0];
assign a_angle1[5:0] = dbg_sw ? dbg_a_angle[5:0] : bos_a_angle1[5:0];
assign a_angle2[5:0] = dbg_sw ? dbg_a_angle2[5:0] : bos_a_angle2[5:0];
assign a_angle3[5:0] = dbg_sw ? dbg_a_angle3[5:0] : bos_a_angle3[5:0];
assign a_angle4[5:0] = dbg_sw ? dbg_a_angle4[5:0] : bos_a_angle4[5:0];
assign v_motion = dbg_sw ? dbg_v_motion : rays_v_motion;
assign a_motion = dbg_sw ? dbg_a_motion : bos_a_motion;

debounce vm(reset, clock_65mhz, ~button3, dbg_v_motion);
debounce hm(reset, clock_65mhz, ~button2, dbg_a_motion);
wire look_up, look_down, look_right, look_left; //outputs of motion calculator
wire [7:0] a_x_center, a_y_center;
//assign a_x_center = 8'b00100010;
//assign a_y_center = 8'b00100101;
wire signed [9:0] a_x_locat, a_y_locat;
//assign a_y_locat = 10'b1100000000; //+512
//assign a_x_locat = 10'b0100000000; //-512
DisplayModule displayer(clock_65mhz, reset,
a_motion, a_angle1[5:0], a_angle2[5:0],
a_angle3[5:0], a_angle4[5:0],
a_x_center, a_y_center,
a_x_locat, a_y_locat, v_motion, v_angle_v[5:0], v_angle_h[5:0],
look_up, look_down, look_right, look_left,
hcount, vcount, hsync, vsync, blank, dhsync, dvsync, dblank, pixel);
wire [1:0] displayChoice;
debounce dis0(reset, clock_65mhz, switch[6], displayChoice[0]);
debounce dis1(reset, clock_65mhz, switch[7], displayChoice[1]);
reg [63:0] hexData = 64'd0;

reg b, hs, vs; //blank, hsync, and vsync vga signals
reg [7:0] red, green, blue; //vga color outputs
wire [63:0] BobbysHexData; //debugging information used to test motion calculator
wire blanker; //output of motion calculator, no motion is allowed for 2 seconds after movement has ended
assign BobbysHexData[63:0] = {{3'b0, picture_we},{3'b0, blanker},
{2'b00, a_angle4[5:0]},{2'b00, a_angle3[5:0]},{2'b00, a_angle2[5:0]},{2'b00, a_angle1[5:0]},{2'b00, a_angle[5:0]},{2'b00, v_angle_h[5:0]},{2'b00, v_angle_v[5:0]}};
always @ (posedge clock_65mhz) begin
  if (~displayChoice[0] && ~displayChoice[1]) begin
    hs <= dhsync;
    vs <= dvsync;
b <= dblank;
hexData[63:0] <= BobbysHexData[63:0];
red <= {8{pixel[2]});
green <= {8{pixel[1]});
blue <= {8{pixel[0]});
end
else if (~displayChoice[0] && displayChoice[1]) begin //10
hs <= Picture_hs;
vs <= Picture_vs;
b <= Picture_b;
hexData[63:0] <= BobbysHexData[63:0];
red <= Picture_Pixel;
green <= Picture_Pixel;
blue <= Picture_Pixel;
end
else begin //11 or 01
hexData[63:0] <= RaysHexData[63:0];
end
end

// VGA Output. In order to meet the setup and hold times of the
// AD7125, we send it ~clock_65mhz.
assign vga_out_red = displayChoice[0] ?  rays_red : red;
assign vga_out_green = displayChoice[0] ?  rays_green : green;
assign vga_out_blue = displayChoice[0] ?  rays_blue : blue;
assign vga_out_sync_b = displayChoice[0] ?  rays_vga_out_sync_b : 1'b1;
assign vga_out_blank_b = displayChoice[0] ?  rays_b : ~b;
assign vga_out_pixel_clock = displayChoice[0] ?  rays_pixel_clock : -clock_65mhz;
assign vga_out_hsync = displayChoice[0] ?  rays_hs : hs;
assign vga_out_vsync = displayChoice[0] ?  rays_vs : vs;

//input buttons
wire move_up, move_down, move_right, move_left;
debounce bup (reset, clock_65mhz, ~button_up, move_up);
debounce bdown (reset, clock_65mhz, ~button_down, move_down);
debounce bright (reset, clock_65mhz, ~button_right, move_right);
debounce bleft (reset, clock_65mhz, ~button_left, move_left);

wire av_priority; //determines whether audio or video motion takes priority
debounce prior(reset, clock_65mhz, switch[5], av_priority);

//outputs of the motion calculator
wire vAp, vAn, vBp, vBn, hAp, hAn, hBp, hBn;
wire moving;
wire cant_move_up, cant_move_down;

MotionCalculator mc(reset, clock_65mhz, manual_override,
move_up, move_down, move_right, move_left,
v_angle_h[5:0], v_angle_v[5:0], v_motion,
a_angle[5:0], a_motion,
av_priority,
cant_move_up, cant_move_down, moving,
vAp, vAn, vBp, vBn,
hAp, hAn, hBp, hBn,
look_up, look_down, look_right, look_left,
v_finished_moving, h_finished_moving, blanker);

//assign beep = a_motion || v_motion; //generate audio beep when there is motion
wire v_enb1, v_enb2, h_enb; //enable signals sent to motor driver (always high)
assign h_enb = 1;
assign v_enb1 = 1;
assign v_enb2 = 1;
wire unsyn_cant_move_up, unsyn_cant_move_down; //unsynchronized signals coming
from the limit switches
assign user3[10:6] = (h_enb, hBn, hBp, hAn, hAp);
assign user3[16:11] = (v_enb1, v_enb2, vBn, vBp, vAn, vAp);
assign unsyn_cant_move_down = user2[0];
assign unsyn_cant_move_up = user2[1];

// debounce cant_move_u(reset, clock_65mhz, unsyn_cant_move_up, cant_move_up);
//debounce cant_move_d(reset, clock_65mhz, unsyn_cant_move_down, cant_move_down);
//remove these signals if you the motor can spin 360 degrees in vertical direction
assign cant_move_up = 0;
assign cant_move_down = 0;

//assign debugging leds:
assign led = ~{manual_override, look_right, look_left, look_up, look_down, reset,
cant_move_up, cant_move_down};
display_16hex hexdisp1(reset, clock_65mhz, hexData[63:0], disp_blank, disp_clock,
disp_ra, disp_ce_b, disp_reset_b, disp_data_out);

//create a special reset for Bo so that the audio calculations are reset after each movement
wire bo_reset = reset || v_finished_moving || h_finished_moving;

//Begin Bo's code
wire reset_timer1, reset_timer2, reset_timer3;
wire expired1, expired2, expired3;
wire [9:0] value1;
assign value1 = 960;
wire [9:0] count1;
wire [3:0] value2;
assign value2 = 10;
wire [3:0] count2;
wire [9:0] value3;
assign value3 = 912;
wire [9:0] count3;
wire reset_timer4;
wire expired4;
wire [10:0] value4;
assign value4 = 1200;
wire [10:0] count4;

//AD7871 driver
wire convst_bar1, cs_bar1, rd_bar1;
wire convst_bar2, cs_bar2, rd_bar2;
wire convst_bar3, cs_bar3, rd_bar3;
wire convst_bar4, cs_bar4, rd_bar4;
wire ready;
wire int_bar1, int_bar2, int_bar3, int_bar4;
wire [2:0] state;
wire [2:0] highstate;
wire [13:0] db1;
wire [13:0] db2;
wire [13:0] db3;
wire [13:0] db4;
wire [1:0] dir1;
wire [1:0] dir2;
wire [9:0] delay1;
wire [9:0] delay2;
wire [2:0] d_state1;
wire [2:0] d_state2;
wire [9:0] d_count1;
wire [9:0] d_count2;
wire reset_possible;
wire [13:0] dividend1;
wire [13:0] dividend2;
wire [9:0] divisor1;
wire [9:0] divisor2;
wire [7:0] d_over_c1;
wire [7:0] d_over_c2;
wire [7:0] slope1;
wire [7:0] slope2;
wire [9:0] t12;
wire [3:0] posangle1;
wire [3:0] posangle2;
wire boblank;

reg [32:0] boblank_count;
always @ (posedge clock_65mhz) begin
  if (bo_reset) boblank_count <= 6500000;
  else if (boblank_count > 0) boblank_count <= boblank_count - 1;
  else boblank_count <= 0;
end
assign boblank = (boblank_count != 0);

timer1 t1(bo_reset, reset_timer1, clock_65mhz, value1, expired1, count1);
timer2 t2(bo_reset, reset_timer2, clock_65mhz, value2, expired2, count2, state);
timer3 t3(bo_reset, reset_timer3, clock_65mhz, value3, expired3, count3);
timer4 t4(bo_reset, reset_timer4, clock_65mhz, value4, expired4, count4);

ad7871_controller adccontrol(bo_reset, clock_65mhz, convst_bar1, convst_bar2, convst_bar3, convst_bar4, cs_bar1, cs_bar2, cs_bar3, cs_bar4, rd_bar1, rd_bar2, rd_bar3, rd_bar4, int_bar1, int_bar2, int_bar3, int_bar4, ready, state, reset_timer1, reset_possible);
differencer diff1(boblank, d_count1, bo_reset, clock_65mhz, expired1, db1, db2, dir1, delay1, d_state1);
differencer diff2(boblank, d_count2, bo_reset, clock_65mhz, expired1, db3, db4, dir2, delay2, d_state2);
slopefinder loc1(boblank, bo_reset, clock_65mhz, delay1, dir1, a_x_center, bos_a_angle1, bos_a_angle2, dividend1, divisor1, d_over_c1, posangle1);
slopefinder loc2(boblank, bo_reset, clock_65mhz, delay2, dir2, a_y_center, bos_a_angle3, bos_a_angle4, dividend2, divisor2, d_over_c2, posangle2);
anglecalc anglecalc1(bo_reset, clock_65mhz, posangle1, posangle2, dir1, dir2, bos_a_angle, bos_a_motion);
assign db1[5:0] = user1[5:0];
assign db1[9:6] = user1[10:7];
assign db1[13:10] = user1[15:12];
assign int_bar1 = user1[31];
assign user3[29] = rd_bar1;
assign user3[28] = cs_bar1;
assign user3[27] = convst_bar1;
assign db2[2:0] = user2[2:0];
assign db2[5:3] = user2[6:4];
assign db2[13:6] = user2[15:8];
assign int_bar2 = user2[31];
assign user3[5] = rd_bar2;
assign user3[4] = cs_bar2;
assign user3[3] = convst_bar2;

assign db3[2:0] = user4[2:0];
assign db3[5:3] = user4[6:4];
assign db3[13:6] = user4[15:8];
assign int_bar3 = user4[31];
assign user3[2] = rd_bar3;
assign user3[1] = cs_bar3;
assign user3[0] = convst_bar3;

assign db4[5:0] = user1[21:16];
assign db4[13:6] = user1[29:22];
assign int_bar4 = user1[30];
assign user3[26] = rd_bar4;
assign user3[25] = cs_bar4;
assign user3[24] = convst_bar4;

//End Bo's code
endmodule

//Rays code:
// Ray Wu: last updated 12-11-05
// Fill video RAM from NTSC decoded video grabbed data
module ntsc_to_ram(reset, clk, fvh, dv, din, vdata, vaddr, vwe, high_contrast, display_one_frame, debug_output, led_output6, logic_output, skippedFrame);
//storage depends on whether we're storing frame A or B
//each frame takes up at most 256x96
//frame A: address => 256x96
//frame B: address => (frame A shift by 256, 96)
input reset;
input clk;
input [2:0] fvh;
input dv;
input [7:0] din;
output vwe;
output [15:0] vaddr;
output [7:0] vdata;
input high_contrast;
input display_one_frame;    //for test purposes (store everything only in mem A)
output [63:0] debug_output; //ray wu debug
output [5:0] led_output6;
output [8:0] logic_output;
output skippedFrame; //if 0, then we are NOT recording (and can do processing)
reg [63:0] debug_output = 0;

//add state control to control where frame will be stored
reg storeInA = 0;
reg storeInB = 0;
reg [1:0] videoMemState; //0: no frames stored, 1: frame A stored, 2: A, B stored (A is history)
//3: B, A stored (B is history)
reg skippedFrame = 0;    //already skipped a frame...time to record again
//store every OTHER frame, if 1, then okay to store
reg [15:0] vaddr = 0;    //need to pick the address depending on storeMemA and storeMemB
reg [2:0] counter_two = 0;
reg just_reset = 0;    //trigger on during reset, tells fsm to start processing only when one frame is finished

//new way of storing frames
//storing frames only need to worry about storeInA and storeInB...
//algorithm...control skippedFrame...when sf is 0, no storing, whenever sf becomes 1, alternate frameStorage
//use skippedFrame, storeInA, and storeInB as before
```verilog
reg [2:0] sf_counter = 0; //counts up to 8
reg history_store = 1; //if last store was A, h_s=1

parameter SKIP_FRAME_TOTAL = 4;
//   parameter   MAX_ROW = 191;
//   parameter   MAX_COL = 255;

// here put the luminance data from the ntsc decoder into the ram
reg [7:0] col = 0;
reg [7:0] row = 0;
reg [7:0] vdata = 0;
//reg    vwe = 0;

//always @ (posedge fvh[1] or posedge reset) //trigger on new frame
always @ (posedge fvh[2] or posedge reset) //trigger on new frame
begin
  if (reset) //changed
    begin
      //memory storage, start with initial state
      storeInA <= 0; //start with storing in A
      storeInB <= 0;
      videoMemState <= 0;
      skippedFrame <= 0;
      //new
      history_store <= 0;
      sf_counter <= 0;
    end
  else
    begin
      //on every change in fvh[2] (happens at 60hz)
      //if (sf_counter == SKIP_FRAME_TOTAL)
      if (sf_counter == SKIP_FRAME_TOTAL)
        begin
          skippedFrame <= 1;
          sf_counter <= 0; //reset the counter
          //We need to switch storing state
          if (history_store) //last store was in A
            begin
              storeInA <= 0;
              storeInB <= 1;
              history_store <= 0; //now the most recent store is in B
            end
          else //last store was in B
            begin
              storeInA <= 1;
              storeInB <= 0;
              history_store <= 1; //now the most recent store is in A
            end
        end
      else //we haven't skipped enough frames
        begin
          skippedFrame <= 0;
          storeInA <= 0; //no storage allowed
          storeInB <= 0;
          sf_counter <= sf_counter + 1;
        end
    end
  end
end

reg old_dv;
reg [2:0] counter_use_pix; //used to obtain every 4th pix
reg use_line = 1; //used to obtain every other line

wire vwe = dv && !fvh[2] && ~old_dv; // if data valid, write it

always @(posedge clk)
begin
  if (reset)
    begin
      counter_use_pix <= 0;
      end
end
```
use_line <= 1;
end
else
begin
old_dv <= dv;
if (!fvh[2])
begin
if (fvh[0])
use_line <= ~use_line;
if ((counter_use_pix == 0) | fvh[1] | fvh[0]) & use_line
begin
col <= fvh[0] ? '8'h00 :
col;
row <= fvh[1] ? '8'h00 :
(!fvh[2] && fvh[0] && (row < 95)) ? (row + 1) : row;
if (high_contrast)
vdata <= (dv && !fvh[2]) ? ((din > 120) ? '8'd255 :
0) : vdata;
/*
vdata <= (dv && !fvh[2]) ? ((din > 200) ? '8'd255 :
((din > 130) ? '8'd225 :
((din > 80) ? '8'd40 :
((din > 30) ? '8'd20 : '8'd0))) : vdata;
*/
else
vdata <= (dv && !fvh[2]) ? din : vdata;
counter_use_pix <= 1;
end
else if (counter_use_pix == 3)
counter_use_pix <= 0;
else
counter_use_pix <= counter_use_pix + 1;
end
end
always @ (posedge clk)
begin
if (!display_one_frame)
begin
if (storeInA) vaddr <= {row, col};
else if (storeInB) vaddr <= {row+96, col};
else vaddr <= 16'b0; //default setting
end
else
vaddr <= {row, col};
end
//test signal for fvh[1]
reg vsync = 0;
always @ (posedge fvh[1])
begin
vsync <= ~vsync;
end
//debug
assign logic_output6[5] = 0;
assign logic_output6[4] = ~skippedFrame;
assign logic_output6[3] = ~storeInA;
assign logic_output6[2] = ~storeInB;
assign logic_output6[1:0] = ~videoMemState[1:0];
//logic analyzer
assign logic_output[8] = reset;
assign logic_output[7:6] = videoMemState[1:0];
assign logic_output[5] = dv;
assign logic_output[4:3] = fvh[2:1];
assign logic_output[2] = storeInA;
assign logic_output[1] = storeInB;
assign logic_output[0] = skippedFrame;
endmodule

// end Rays code
module debounce (reset, clock_65mhz, noisy, clean);
  input reset, clock_65mhz, noisy;
  output clean;
  parameter DELAY = 648000; //~0.01 seconds on a 65mhz clock
  reg [20:0] count;
  reg new, clean;
  always @(posedge clock_65mhz)
    if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
    else if (noisy != new) begin new <= noisy; count <= 0; end
    else if (count == 648000) clean <= new;
    else count <= count+1;
endmodule

module synchronize(clk,in,out);
  parameter NSYNC = 2; // number of sync flops. must be >= 2
  input clk;
  input in;
  output out;
  reg [NSYNC-2:0] sync;
  reg out;
  always @(posedge clk)
    begin
      {out,sync} <= {sync[NSYNC-2:0],in};
    end
endmodule
module display_16hex (reset, clk, data_in, disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b, disp_data_out);

input reset, clk;    // clock and reset (active high reset)
input [63:0] data_in;  // 16 hex nibbles to display
parameter CLK_SPEED = 65;

output disp_blank, disp_clock, disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

always @(posedge clk)
begin
  disp_data_out <= data_in[63:56]; // First nibble
  disp_data_out <= data_in[55:48]; // Second nibble
  disp_data_out <= data_in[47:40]; // Third nibble
  disp_data_out <= data_in[39:32]; // Fourth nibble
  disp_data_out <= data_in[31:24]; // Fifth nibble
  disp_data_out <= data_in[23:16]; // Sixth nibble
  disp_data_out <= data_in[15:8];  // Seventh nibble
  disp_data_out <= data_in[7:0];   // Eighth nibble
end

assign disp_blank = !!reset;
assign disp_clock = ~~clk;
assign disp_data_out = ~~disp_data_out;
wire clock_tick = ((count<2*CLK_SPEED) ? 1 : 0);

endmodule
wire  clock_tick = clock & ~old_clock;

// Display State Machine

reg [7:0] state;  // FSM state
reg [9:0] dot_index;  // index to current dot being clocked out
reg [31:0] control;  // control register
reg [3:0] char_index;  // index of current character
reg [39:0] dots;  // dots for a single digit
reg [3:0] nibble;  // hex nibble of current character
reg [63:0] data;

assign disp_blank = 1'b0; // low <= not blanked

always @(posedge clk)
if (clock_tick)
begin
  if (dreset)
  begin
    state <= 0;
    dot_index <= 0;
    control <= 32'h7F7F7F7F;
  end
  else
    casex (state)
    8'h00:
      begin
        // Reset displays
        disp_data_out <= 1'b0;
        disp_rs <= 1'b0; // dot register
        disp_ce_b <= 1'b1;
        disp_reset_b <= 1'b0;
        dot_index <= 0;
        state <= state+1;
      end
    8'h01:
      begin
        // End reset
        disp_reset_b <= 1'b1;
        state <= state+1;
      end
    8'h02:
      begin
        // Initialize dot register (set all dots to zero)
        disp_ce_b <= 1'b0;
        disp_data_out <= 1'b0; // dot_index[0];
        if (dot_index == 639)
          state <= state+1;
        else
          dot_index <= dot_index+1;
      end
    8'h03:
      begin
        // Latch dot data
        disp_ce_b <= 1'b1;
        dot_index <= 31;  // re-purpose to init ctrl reg
        state <= state+1;
      end
    8'h04:
      begin
        // Setup the control register
        disp_rs <= 1'b1; // Select the control register
        disp_ce_b <= 1'b0;
      end
  end
disp_data_out <= control[31];
control <= {control[30:0], 1'b0}; // shift left
if (dot_index == 0)
    state <= state+1;
else
    dot_index <= dot_index-1;
end

8'h05:
begin
    // Latch the control register data / dot data
    disp_ce_b <= 1'b1;
dot_index <= 39;  // init for single char
char_index <= 15;  // start with MS char
data <= data_in;
state <= state+1;
end

8'h06:
begin
    // Load the user's dot data into the dot reg, char by char
    disp_rs <= 1'b0;    // Select the dot register
disp_ce_b <= 1'b0;
disp_data_out <= dots[dot_index]; // dot data from msb
if (dot_index == 0)
    if (char_index == 0)
        state <= 5;   // all done, latch data
    else
        begin
            char_index <= char_index - 1; // goto next char
data <= data_in;
dot_index <= 39;
        end
else
    begin
        dot_index <= dot_index-1; // else loop thru all dots
    end
endcase // casex(state)
end

always @(data or char_index)
case (char_index)
  4'h0:    nibble <= data[3:0];
  4'h1:    nibble <= data[7:4];
  4'h2:    nibble <= data[11:8];
  4'h3:    nibble <= data[15:12];
  4'h4:    nibble <= data[19:16];
  4'h5:    nibble <= data[23:20];
  4'h6:    nibble <= data[27:24];
  4'h7:    nibble <= data[31:28];
  4'h8:    nibble <= data[35:32];
  4'h9:    nibble <= data[39:36];
  4'hA:    nibble <= data[43:40];
  4'hB:    nibble <= data[47:44];
  4'hC:    nibble <= data[51:48];
  4'hD:    nibble <= data[55:52];
  4'hE:    nibble <= data[59:56];
  4'hF:    nibble <= data[63:60];
endcase

always @(nibble)
case (nibble)
  4'h0: dots <= 40'b00111110_01010001_01001001_01000101_00111110;
  4'h1: dots <= 40'b00000000_01000010_01111111_01000000_00000000;
  4'h2: dots <= 40'b01100010_01001001_01001001_01001001;
  4'h3: dots <= 40'b00101001_01001001_01001001_01001001;
  4'h4: dots <= 40'b01011000_01001001_01001001;
  4'h5: dots <= 40'b00101001_01001001;
  4'h6: dots <= 40'b00101001;
  4'h7: dots <= 40'b00101001;
  4'h8: dots <= 40'b00101001;
endcase
```verilog
4'h9: dots <= 40'b00000110_01001001_01001001_00101001_00011110;
4'hA: dots <= 40'b01111110_00001001_00001001_00001001_01111110;
4'hB: dots <= 40'b00111111_01001001_01001001_01001001_00110110;
4'hC: dots <= 40'b00111110_01000001_01000001_01000001_00100010;
4'hD: dots <= 40'b00111111_01000001_01000001_01000001_00111110;
4'hE: dots <= 40'b01111111_01001001_01001001_01001001_01000001;
4'hF: dots <= 40'b01111111_00001001_00001001_00001001_00000001;
endcase
endmodule
```
module DisplayModule(vclock, reset, 
a_motion, a_angle,
a_angle2, a_angle3, a_angle4, 
a_x_center, a_y_center,
a_locat, a_locat,
v_motion, v_angle_v, v_angle_h, 
up, down, right, left, 
hcount, vcount, hsync, vsync, blank, 
dhsync, dvsync, dbank, 
pixel);
input vclock;  // 65MHz clock
input reset;  // 1 to initialize module
input a_motion, v_motion; //signals indicating audio or v_motion was detected
input [5:0] a_angle, v_angle_v, v_angle_h; //the angles to display on the motion compasses
input [5:0] a_angle2, a_angle3, a_angle4; //all possible locations of the sound
input [7:0] a_x_center; //center of audio angle from 0-256
input [7:0] a_y_center;
input [9:0] a_x_locat, a_y_locat; //10 bit location of audio sound
input up, down, left, right; //displays current motion of motor
input [10:0] hcount;  // horizontal index of current pixel (0..1023)
input [9:0] vcount; // vertical index of current pixel (0..767)
input hsync;  // XVGA horizontal sync signal (active low)
input vsync;  // XVGA vertical sync signal (active low)
input blank;  // XVGA blanking (1 means output black pixel)
output dhsync; // output horizontal sync
output dvsync; // output vertical sync
output dbank; // output blanking
output [2:0] pixel; // current pixel rgb

parameter screen_size_x = 1024;
parameter screen_size_y = 768;
assign dhsync = hsync;
assign dvsync = vsync;
assign dbank = blank;

//draw vertical and horizontal separators to split the screen into quadrants
parameter separator_width = 4;
wire [2:0] horiz_sep_pixel, vert_sep_pixel, separator_pixels;
rectangle h_separator(11'd0,(screen_size_y - separator_width)/2,hcount,vcount, 
horiz_sep_pixel);
rectangle v_separator((screen_size_x - separator_width)/2,10'd0,hcount,vcount, 
vert_sep_pixel);
defparam h_separator.WIDTH = screen_size_x; //full length of screen 
defparam h_separator.HEIGHT = separator_width;
defparam h_separator.COLOR = 3'b110; //yellow

defparam v_separator.WIDTH = separator_width; 
defparam v_separator.HEIGHT = screen_size_y; //full height of screen

defparam v_separator.COLOR = 3'b110; //yellow
assign separator_pixels = horiz_sep_pixel + vert_sep_pixel;

//upper left Quadrant
//draw debugging module for audio (2 microphones and the possible locations of motion)
wire [10:0] real_a_x_center;
assign real_a_x_center = {3'b000, a_x_center + screen_size_x / 8};
wire [2:0] mic1_pixel, mic2_pixel, center_pixel, line1_pixel, line2_pixel;
rectangle mic1(screen_size_x/8, screen_size_y/4, hcount, vcount, mic1_pixel);
defparam mic1.WIDTH = 20;
defparam mic1.HEIGHT = 20;
defparam mic1.COLOR = 3'b101;
rectangle mic2(screen_size_x * 3/8, screen_size_y/4, hcount, vcount, mic2_pixel);
defparam mic2.WIDTH = 20;
defparam mic2.HEIGHT = 20;
defparam mic2.COLOR = 3'b101;
rectangle center(real_a_x_center, screen_size_y/4 + 5, hcount, vcount, center_pixel);
defparam center.WIDTH = 10;
defparam center.HEIGHT = 10;
defparam center.COLOR = 3'b101;
wire [10:0] end_x1_1, end_x1_2, end_x1_3, end_x2_1, end_x2_2, end_x2_3;
wire [9:0] end_y1_1, end_y1_2, end_y1_3, end_y2_1, end_y2_2, end_y2_3;
wire [2:0] line1_1_pixel, line1_2_pixel, line1_3_pixel, line2_1_pixel, line2_2_pixel, line2_3_pixel;

//draw the lines by calculating the end_pt of each line with various increasing radii
  calc_end_pt liner1_1(a_angle[5:0], real_a_x_center, screen_size_y/4+5, 32, end_x1_1, end_y1_1);
  calc_end_pt liner1_2(a_angle[5:0], real_a_x_center, screen_size_y/4+5, 64, end_x1_2, end_y1_2);
  calc_end_pt liner1_3(a_angle[5:0], real_a_x_center, screen_size_y/4+5, 96, end_x1_3, end_y1_3);
  calc_end_pt liner2_1(a_angle[5:0], real_a_x_center, screen_size_y/4+5, 32, end_x2_1, end_y2_1);
  calc_end_pt liner2_2(a_angle[5:0], real_a_x_center, screen_size_y/4+5, 64, end_x2_2, end_y2_2);
  calc_end_pt liner2_3(a_angle[5:0], real_a_x_center, screen_size_y/4+5, 96, end_x2_3, end_y2_3);

//draw rectangles along the line
  rectangle line1_1(end_x1_1, end_y1_1, hcount, vcount, line1_1_pix);
defparam line1_1.WIDTH = 10;
defparam line1_1.HEIGHT = 10;
  rectangle line1_2(end_x1_2, end_y1_2, hcount, vcount, line1_2_pix);
defparam line1_2.WIDTH = 10;
defparam line1_2.HEIGHT = 10;
  rectangle line1_3(end_x1_3, end_y1_3, hcount, vcount, line1_3_pix);
defparam line1_3.WIDTH = 10;
defparam line1_3.HEIGHT = 10;
  rectangle line2_1(end_x2_1, end_y2_1, hcount, vcount, line2_1_pix);
defparam line2_1.WIDTH = 10;
defparam line2_1.HEIGHT = 10;
  rectangle line2_2(end_x2_2, end_y2_2, hcount, vcount, line2_2_pix);
defparam line2_2.WIDTH = 10;
defparam line2_2.HEIGHT = 10;
  rectangle line2_3(end_x2_3, end_y2_3, hcount, vcount, line2_3_pix);
defparam line2_3.WIDTH = 10;
defparam line2_3.HEIGHT = 10;

assign line1_pixel = line1_1_pix + line1_2_pix + line1_3_pix;
assign line2_pixel = line2_1_pix + line2_2_pix + line2_3_pix;
wire [10:0] real_a_y_center;
assign real_a_y_center[10:0] = a_y_center + screen_size_y / 8;
wire [2:0] mic3_pixel, mic4_pixel, center_y_pixel, line3_pixel, line4_pixel;
rectangle mic3(screen_size_x/4, screen_size_y/8, hcount, vcount, mic3_pixel);
defparam mic3.WIDTH = 20;
defparam mic3.HEIGHT = 20;
defparam mic3.COLOR = 3'b101;
rectangle mic4(screen_size_x/4, screen_size_y * 3/8, hcount, vcount, mic4_pixel);
defparam mic4.WIDTH = 20;
defparam mic4.HEIGHT = 20;
defparam mic4.COLOR = 3'b101;
rectangle center_y(screen_size_x/4 + 5, real_a_y_center, hcount, vcount, center_y_pixel);
defparam center_y.WIDTH = 10;
defparam center_y.HEIGHT = 10;
defparam center_y.COLOR = 3'b101;
wire [10:0] end_x3_1, end_x3_2, end_x3_3, end_x4_1, end_x4_2, end_x4_3;
wire [9:0] end_y3_1, end_y3_2, end_y3_3, end_y4_1, end_y4_2, end_y4_3;
wire [2:0] line3_1_pix, line3_2_pix, line3_3_pix, line4_1_pix, line4_2_pix, line4_3_pix;

//rotate vertical angles by 90 degrees (mod 36)
wire [5:0] rotated_a3, rotated_a4;
assign rotated_a3[5:0] = (a_angle3[5:0] > 26) ? (9 - (35 - a_angle3[5:0])) : (a_angle3[5:0] + 9);
assign rotated_a4[5:0] = (a_angle4[5:0] > 26) ? (9 - (35 - a_angle4[5:0])) : (a_angle4[5:0] + 9);

//draw the lines by calculating the end_pt of of each line with various increasing radii
/*calc_end_pt liner3_1(a_angle3[5:0], screen_size_x/4 + 5, real_a_y_center, 32, end_x3_1, end_y3_1);
calc_end_pt liner3_2(a_angle3[5:0], screen_size_x/4 + 5, real_a_y_center, 64, end_x3_2, end_y3_2);
calc_end_pt liner3_3(a_angle3[5:0], screen_size_x/4 + 5, real_a_y_center, 96, end_x3_3, end_y3_3);
calc_end_pt liner4_1(a_angle4[5:0], screen_size_x/4 + 5, real_a_y_center, 32, end_x4_1, end_y4_1);
calc_end_pt liner4_2(a_angle4[5:0], screen_size_x/4 + 5, real_a_y_center, 64, end_x4_2, end_y4_2);
calc_end_pt liner4_3(a_angle4[5:0], screen_size_x/4 + 5, real_a_y_center, 96, end_x4_3, end_y4_3);*/
calc_end_pt liner3_1(rotated_a3[5:0], screen_size_x/4 + 5, real_a_y_center, 32, end_x3_1, end_y3_1);
calc_end_pt liner3_2(rotated_a3[5:0], screen_size_x/4 + 5, real_a_y_center, 64, end_x3_2, end_y3_2);
calc_end_pt liner3_3(rotated_a3[5:0], screen_size_x/4 + 5, real_a_y_center, 96, end_x3_3, end_y3_3);
calc_end_pt liner4_1(rotated_a4[5:0], screen_size_x/4 + 5, real_a_y_center, 32, end_x4_1, end_y4_1);
calc_end_pt liner4_2(rotated_a4[5:0], screen_size_x/4 + 5, real_a_y_center, 64, end_x4_2, end_y4_2);
calc_end_pt liner4_3(rotated_a4[5:0], screen_size_x/4 + 5, real_a_y_center, 96, end_x4_3, end_y4_3);

//draw rectangles along the line
rectangle line3_1(end_x3_1, end_y3_1, hcount, vcount, line3_1_pix);
defparam line3_1.WIDTH = 10;
defparam line3_1.HEIGHT = 10;
rectangle line3_2(end_x3_2, end_y3_2, hcount, vcount, line3_2_pix);
defparam line3_2.WIDTH = 10;
defparam line3_2.HEIGHT = 10;
rectangle line3_3(end_x3_3, end_y3_3, hcount, vcount, line3_3_pix);
defparam line3_3.WIDTH = 10;
defparam line3_3.HEIGHT = 10;
rectangle line4_1(end_x4_1, end_y4_1, hcount, vcount, line4_1_pix);
defparam line4_1.WIDTH = 10;
defparam line4_1.HEIGHT = 10;
rectangle line4_2(end_x4_2, end_y4_2, hcount, vcount, line4_2_pix);
defparam line4_2.WIDTH = 10;
defparam line4_2.HEIGHT = 10;
rectangle line4_3(end_x4_3, end_y4_3, hcount, vcount, line4_3_pix);
defparam line4_3.WIDTH = 10;
defparam line4_3.HEIGHT = 10;
//sum up all the pixels
assign line3_pixel = line3_1_pix + line3_2_pix + line3_3_pix;
assign line4_pixel = line4_1_pix + line4_2_pix + line4_3_pix;
wire [2:0] audio_debug_pixels;
assign audio_debug_pixels = mic1_pixel + mic2_pixel + mic3_pixel + mic4_pixel +
center_pixel + center_y_pixel +
line1_pixel + line2_pixel + line3_pixel +
line4_pixel;

//upper right quadrant: x-y grid displaying location of sound
//draw quadrant lines
wire [2:0] x_axis_pixel, y_axis_pixel;
rectangle x_axis(screen_size_x * 9/16, (screen_size_y / 4), hcount, vcount, x_axis_pixel);
defparam x_axis.WIDTH = 384;
defparam x_axis.HEIGHT = 5;
rectangle y_axis(screen_size_x * 3/4, 0, hcount, vcount, y_axis_pixel);
defparam y_axis.WIDTH = 5;
defparam y_axis.HEIGHT = 384;

wire signed [10:0] a_x_location = (a_x_locat * 3/8) + screen_size_x*9/16 + 192;
wire signed [10:0] a_y_location = 192 - (a_y_locat * 3/8);
wire [2:0] a_spot_pixel;
rectangle a_spot(a_x_location[10:0], a_y_location[9:0], hcount, vcount, a_spot_pixel);
defparam a_spot.WIDTH = 10;
defparam a_spot.HEIGHT = 10;
defparam a_spot.COLOR = 3'b101;

wire [10:0] a_locat_pixels = x_axis_pixel + y_axis_pixel + a_spot_pixel;

//bottom left Quadrant
wire [2:0] motion_pixels;
//draw up-down-right-left arrows and a circle around them
wire [2:0] up_pixel, up_pixel_on, down_pixel, down_pixel_on, right_pixel,
right_pixel_on, left_pixel, left_pixel_on, circle_pix;
rectangle upArrow(screen_size_x/4-5, screen_size_y * 3/4-105, hcount, vcount,
up_pixel_on);
defparam upArrow.WIDTH = 10;
defparam upArrow.HEIGHT = 100;
defparam upArrow.COLOR = 3'b101;
rectangle downArrow(screen_size_x/4-5, screen_size_y * 3/4+5, hcount, vcount,
down_pixel_on);
defparam downArrow.WIDTH = 10;
defparam downArrow.HEIGHT = 100;
defparam downArrow.COLOR = 3'b101;
rectangle leftArrow(screen_size_x/4 - 105, screen_size_y * 3/4 - 5, hcount, vcount,
left_pixel_on);
defparam leftArrow.WIDTH = 10;
defparam leftArrow.HEIGHT = 10;
defparam leftArrow.COLOR = 3'b101;
rectangle rightArrow(screen_size_x/4 + 5, screen_size_y * 3/4 - 5, hcount, vcount,
right_pixel_on);
defparam rightArrow.WIDTH = 10;
defparam rightArrow.HEIGHT = 10;
defparam rightArrow.COLOR = 3'b101;
compass circ(screen_size_x/4, screen_size_y * 3/4, 133, 5, 1'b0, 6'd0, hcount, vcount,
circle_pix);
defparam circ.COLOR = 3'b011;

//we only draw the pixels when up, down, left, right is active high
assign up_pixel = up ? up_pixel_on : 3'b000;
assign down_pixel = down ? down_pixel_on : 3'b000;
assign right_pixel = right ? right_pixel_on : 3'b000;
assign left_pixel = left ? left_pixel_on : 3'b000;
assign motion_pixels = up_pixel + down_pixel + left_pixel + right_pixel + circle_pix;

//Bottom Right Quadrant
wire [2:0] compass_pixels;
//draw a compass for horizontal audio, horizontal video, and vertical video motion by
drawing two concentric circles of differing color in bottom right quadrant
//uncomment compass declarations if you only want to see the angles when motion is
detected
parameter motion_compass_radius = 11'b00001000000; //64 pixel compass radius
wire [2:0] h_audio_compass_pixel, h_video_compass_pixel, v_video_compass_pixel;
wire [2:0] h_audio_compass_pixel2;
compass h_video_compass(screen_size_x * 5/8, screen_size_y *5/8,
motion_compass_radius, separator_width, v_motion, v_angle_h[5:0], hcount, vcount,
h_video_compass_pixel);
compass h_video_compass(screen_size_x * 5/8, screen_size_y *5/8,
motion_compass_radius, separator_width, 1'b1, v_angle_h[5:0], hcount, vcount,
h_video_compass_pixel2);
defparam h_video_compass.COLOR = 3'b111; //white
\_compass v_video_compass(screen_size_x * 7/8, screen_size_y *5/8,
motion_compass_radius, separator_width, v_motion, v_angle_v[5:0], hcount, vcount,
v_video_compass_pixel);
\_compass v_video_compass(screen_size_x * 7/8, screen_size_y *5/8,
motion_compass_radius, separator_width, 1'b1, v_angle_v[5:0], hcount, vcount,
v_video_compass_pixel);
\_defparam v_video_compass.COLOR = 3'b111; //white
\_compass h_audio_compass(screen_size_x * 5/8, screen_size_y *7/8,
motion_compass_radius, separator_width, a_motion, a_angle[5:0], hcount, vcount,
h_audio_compass_pixel);
\_compass h_audio_compass(screen_size_x * 5/8, screen_size_y *7/8,
motion_compass_radius, separator_width, 1'b1, a_angle[5:0], hcount, vcount,
h_audio_compass_pixel);
\_defparam h_audio_compass.COLOR = 3'b100; //red
//second "special" audio compass
\_compass h_audio_compass2(screen_size_x * 7/8, screen_size_y *7/8,
motion_compass_radius, separator_width, a_motion, a_angle2[5:0], hcount, vcount,
h_audio_compass_pixel2);
\_compass h_audio_compass2(screen_size_x * 7/8, screen_size_y *7/8,
motion_compass_radius, separator_width, 1'b1, a_angle2[5:0], hcount, vcount,
h_audio_compass_pixel2);
\_defparam h_audio_compass2.COLOR = 3'b100; //red
//draw labels:
wire [2:0] label_pixels;
//draw a V on the screen to label video compasses
\_wire [2:0] v1_pix, v2_pix, v3_pix;
\_rectangle v1(screen_size_x*17/32, screen_size_y * 5/8, hcount, vcount, v1_pix);
\_defparam v1.WIDTH = 5;
\_defparam v1.HEIGHT = 20;
\_defparam v1.COLOR = 3'b111; //white
\_rectangle v2(screen_size_x*17/32 + 5, screen_size_y * 5/8 + 5, hcount, vcount,
v2_pix);
\_defparam v2.WIDTH = 5;
\_defparam v2.HEIGHT = 5;
\_defparam v2.COLOR = 3'b111;
\_rectangle v3(screen_size_x*17/32 + 10, screen_size_y * 5/8, hcount, vcount, v3_pix);
\_defparam v3.WIDTH = 5;
\_defparam v3.HEIGHT = 20;
\_defparam v3.COLOR = 3'b111;
//draw an A on the screen to label audio compasses
\_wire [2:0] a1_pix, a2_pix, a3_pix, a4_pix;
\_rectangle a1(screen_size_x*17/32, screen_size_y * 7/8, hcount, vcount, a1_pix);
\_defparam a1.WIDTH = 5;
\_defparam a1.HEIGHT = 20;
\_defparam a1.COLOR = 3'b100; //red
\_rectangle a2(screen_size_x*17/32 + 5, screen_size_y * 7/8, hcount, vcount, a2_pix);
\_defparam a2.WIDTH = 5;
\_defparam a2.HEIGHT = 5;
\_defparam a2.COLOR = 3'b100; //red
\_rectangle a3(screen_size_x*17/32 + 10, screen_size_y * 7/8, hcount, vcount, a3_pix);
\_defparam a3.WIDTH = 5;
\_defparam a3.HEIGHT = 20;
\_defparam a3.COLOR = 3'b100; //red
\_rectangle a4(screen_size_x*17/32 + 5, screen_size_y * 7/8 + 10, hcount, vcount,
a4_pix);
\_defparam a4.WIDTH = 5;
\_defparam a4.HEIGHT = 5;
\_defparam a4.COLOR = 3'b100; //red
//draw an H on the screen to label horizontal compass
\_wire [2:0] h1_pix, h2_pix, h3_pix;
\_rectangle h1(screen_size_x*5/8 - 5, screen_size_y/2 + 10, hcount, vcount, h1_pix);
\_defparam h1.WIDTH = 5;
\_defparam h1.HEIGHT = 20;
\_defparam h1.COLOR = 3'b111;
\_rectangle h2(screen_size_x*5/8, screen_size_y/2+17, hcount, vcount, h2_pix);
\_defparam h2.WIDTH = 5;
defparam h2.HEIGHT = 5;
defparam h2.COLOR = 3'b111;
rectangle h3(screen_size_x*5/8+5, screen_size_y/2 + 10, hcount, vcount, h3_pix);
defparam h3.WIDTH = 5;
defparam h3.HEIGHT = 20;
defparam h3.COLOR = 3'b111;

//draw a V on the screen to label vertical compass
wire [2:0] vv1_pix, vv2_pix, vv3_pix;
rectangle vv1(screen_size_x*7/8 - 5, screen_size_y/2 + 10, hcount, vcount, vv1_pix);
defparam vv1.WIDTH = 5;
defparam vv1.HEIGHT = 20;
defparam vv1.COLOR = 3'b111; //white
rectangle vv2(screen_size_x*7/8, screen_size_y/2 + 25, hcount, vcount, vv2_pix);
defparam vv2.WIDTH = 5;
defparam vv2.HEIGHT = 5;
defparam vv2.COLOR = 3'b111;
rectangle vv3(screen_size_x*7/8 + 5, screen_size_y/2 + 10, hcount, vcount, vv3_pix);
defparam vv3.WIDTH = 5;
defparam vv3.HEIGHT = 20;
defparam vv3.COLOR = 3'b111;

assign label_pixels = v1_pix + v2_pix + v3_pix +
a1_pix + a2_pix + a3_pix + a4_pix +
h1_pix + h2_pix + h3_pix +
v1_pix + vv2_pix + vv3_pix;

assign compass_pixels = h_audio_compass_pixel + //h_audio_compass_pixel2 +
h_video_compass_pixel + v_video_compass_pixel +
label_pixels;

//sum up all the pixels of each individual quadrant
assign pixel = separator_pixels +
motion_pixels +
compass_pixels +
audio_debug_pixels +
a_locat_pixels;
endmodule

module rectangle(x,y,hcount,vcount, pixel);
//a module that will assign pixel to COLOR if the coordinate pair (hcount,vcount)
is within the square
//that starts at (x,y) with width WIDTH and height HEIGHT
parameter WIDTH = 64; //64 pixels wide
parameter HEIGHT = 64; //64 pixels tall
parameter COLOR = 3'b111; //white

input [10:0] x,hcount;
input [9:0] y,vcount;
output [2:0] pixel;

reg [2:0] pixel;

always @(x or y or hcount or vcount) begin
    if((hcount >= x && hcount < (x+WIDTH)) &&
        (vcount >= y && vcount < (y+HEIGHT))) begin
        pixel = COLOR;
    end
    else pixel = 0;
end //end always
endmodule

module circle(x,y,r,hcount,vcount,pixel);
//a module that will assign pixel to COLOR if the coordinate pair (hcount, vcount)
is within the circle
//centered at (x,y) with radius r
//to avoid noise this module could be rewritten to draw a sprite from ROM but
since this is just a debugging module
//it may not be worth the effort

parameter COLOR = 3'b111; //white

input [10:0] x, hcount; //x-coordinate of center of the circle, and the current x-coord of the xvga signal
input [9:0] y, vcount; //y-coordinate of center of the circle, and the current y-coord of the xvga signal
input [10:0] r; //radius of the circle
output [2:0] pixel; //output pixel

reg [2:0] pixel;
always @(x or y or r or hcount or vcount) begin
    //check to see if x^2 + y^2 <= r^2, if it is then we are inside the circle
    //to speed things up and avoid noise first check to see if its inside the box surrounding the circle
    if (hcount > (x-r) && hcount < (x+r) && vcount > (y-r) && vcount < (y+r))
    begin
        if (x > hcount) begin
            if (y > vcount) begin
                if ((x - hcount) * (x - hcount) + (y - vcount) * (y - vcount) <= r*r) begin
                    pixel = COLOR;
                end
            end
            else if((x - hcount) * (x - hcount) + (vcount - y) * (vcount - y) <= r*r) begin
                pixel = COLOR;
            end
            else pixel = 0;
        end
        else if (y > vcount) begin
            if ((hcount - x) * (hcount - x) + (y - vcount) * (y - vcount) <= r*r) begin
                pixel = COLOR;
            end
            else if((hcount - x) * (hcount - x) + (vcount - y) * (vcount - y) <= r*r) begin
                pixel = COLOR;
            end
            else pixel = 0;
        end
        else pixel = 0;
    end
endmodule

module compass(x,y,r,w,valid_angle,angle,hcount,vcount,pixel);
input [10:0] x; //x coordinate of the center of the compass
input [9:0] y; //y coordinate of the center of the compass
input [10:0] r; //radius of compass
input [2:0] w; //width of outer compass display
input [5:0] angle; //angle of compass needle
input valid_angle; //determines whether angle is valid
input [10:0] hcount;
input [9:0] vcount;
output [2:0] pixel;
parameter COLOR = 3'b111; //white
wire [2:0] pixel_out, pixel_in;

//make two concentric circles with varying colors to display the outer ring
circle outer_circle(x, y, r, w, hcount, vcount, pixel_out);
defparam outer_circle.COLOR = COLOR; //3'b111; //white

circle inner_circle(x, y, r - w, hcount, vcount, pixel_in);
defparam inner_circle.COLOR = 8 - COLOR; //when added to the outer color
circle this will be black

wire [2:0] needle_pixel;
wire [10:0] end_pt_x;
wire [9:0] end_pt_y;

//compute start and end points of the needle
//note that this is computed where 0 degrees at the positive y axis, otherwise

sin/cos would be flopped

//end_pt_x = x + r*sin(10*angle)
//end_pt_y = y + r*cos(10*angle)
calc_end_pt calker(angle[5:0], x, y, r-2*w, end_pt_x, end_pt_y);

wire [2:0] invalid_needle_pixel;
rectangle needle(end_pt_x,end_pt_y, hcount, vcount, invalid_needle_pixel);
defparam needle.WIDTH = 16;
defparam needle.HEIGHT = 16;
defparam needle.COLOR = 3'b001;  //blue
assign needle_pixel = valid_angle ? invalid_needle_pixel : 3'b000;
assign pixel = pixel_out + pixel_in + needle_pixel;
endmodule

module calc_end_pt(angle, x, y, r, end_pt_x, end_pt_y);
input [5:0] angle;
input [10:0] x;
input [9:0] y;
input [10:0] r;
output [10:0] end_pt_x;
output [9:0] end_pt_y;

//a better implementation would use coregen's trig function but this is just a
quick dumb module for gui display
//note that the angles come in increments of 10 degrees from 0-35

reg [10:0] end_pt_x = 0;
reg [9:0] end_pt_y = 0;

always @ (x or y or angle or r) begin
  case (angle[5:0])
    6'd0: begin
      end_pt_x = x;
      end_pt_y = y - r;
    end
    6'd1: begin
      end_pt_x = x + (r * 6/32);
      end_pt_y = y - (r * 31/32);
    end
    6'd2: begin
      end_pt_x = x + (r * 11/32);
      end_pt_y = y - (r * 30/32);
    end
    6'd3: begin
      end_pt_x = x + r/2;
      end_pt_y = y - (r * 28/32);
    end
    6'd4: begin
      end_pt_x = x + (r * 21/32);
      end_pt_y = y - (r * 25/32);
    end
    6'd5: begin
      end_pt_x = x + (r * 25/32);
      end_pt_y = y - (r * 21/32);
    end
    6'd6: begin
      end_pt_x = x + (r * 28/32);
      end_pt_y = y - r/2;
    end
  end
endmodule
end pt_x = x + (r * 30/32);
end pt_y = y - (r * 11/32);
end
6’d8: begin
end pt_x = x + (r * 31/32);
end pt_y = y - (r * 6/32);
end
6’d9: begin
end pt_x = x + r;
end pt_y = y;
end
6’d10: begin
end pt_x = x + (r * 31/32);
end pt_y = y + (r * 6/32);
end
6’d11: begin
end pt_x = x + (r * 30/32);
end pt_y = y + (r * 11/32);
end
6’d12: begin
end pt_x = x + (r * 28/32);
end pt_y = y + r/2;
end
6’d13: begin
end pt_x = x + (r * 25/32);
end pt_y = y + (r * 21/32);
end
6’d14: begin
end pt_x = x + (r * 21/32);
end pt_y = y + (r * 25/32);
end
6’d15: begin
end pt_x = x + r/2;
end pt_y = y + (r * 28/32);
end
6’d16: begin
end pt_x = x + (r * 11/32);
end pt_y = y + (r * 30/32);
end
6’d17: begin
end pt_x = x + (r * 6/32);
end pt_y = y + (r * 31/32);
end
6’d18: begin
end pt_x = x;
end pt_y = y + r;
end
6’d19: begin
end pt_x = x - (r * 6/32);
end pt_y = y + (r * 31/32);
end
6’d20: begin
end pt_x = x - (r * 11/32);
end pt_y = y + (r * 30/32);
end
6’d21: begin
end pt_x = x - r/2;
end pt_y = y + (r * 28/32);
end
6’d22: begin
end pt_x = x - (r * 21/32);
end pt_y = y + (r * 25/32);
end
6’d23: begin
end pt_x = x - (r * 25/32);
end pt_y = y + (r * 21/32);
end
6’d24: begin
end pt_x = x - (r * 28/32);
end pt_y = y + r/2;
end
6'd25: begin
  end_pt_x = x - (r * 30/32);
  end_pt_y = y + (r * 11/32);
end
6'd26: begin
  end_pt_x = x - (r * 31/32);
  end_pt_y = y + (r * 6/32);
end
6'd27: begin
  end_pt_x = x - r;
  end_pt_y = y;
end
6'd28: begin
  end_pt_x = x - (r * 31/32);
  end_pt_y = y - (r * 6/32);
end
6'd29: begin
  end_pt_x = x - (r * 30/32);
  end_pt_y = y - (r * 11/32);
end
6'd30: begin
  end_pt_x = x - (r * 28/32);
  end_pt_y = y - r/2;
end
6'd31: begin
  end_pt_x = x - (r * 25/32);
  end_pt_y = y - (r * 21/32);
end
6'd32: begin
  end_pt_x = x - (r * 21/32);
  end_pt_y = y - (r * 25/32);
end
6'd33: begin
  end_pt_x = x - r/2;
  end_pt_y = y - (r * 28/32);
end
6'd34: begin
  end_pt_x = x - (r * 11/32);
  end_pt_y = y - (r * 30/32);
end
6'd35: begin
  end_pt_x = x - (r * 6/32);
  end_pt_y = y - (r * 31/32);
end
default: begin
  end_pt_x = x;
  end_pt_y = y - r;
endcasenendmodule
module MotionCalculator(reset, clk, Manual_Override, m_move_up, m_move_down, m_move_right, m_move_left, v_angle_h, v_angle_v, v_motion, a_angle, a_motion, priority, cant_move_up, cant_move_down, moving, vIAp, vIAn, vIBp, vIBn, hIAp, hIAn, hIBp, hIBn, look_up, look_down, look_right, look_left, v_expired, h_expired, blank);

// parameter motor_clk_speed = 100; //frequency of steps
// parameter motor_speed = 10; //degrees per second
// parameter MOTOR_SPEED_FACTOR = 2; //this multiplies the angle to determine how long to hold the motor drive signals high
// parameter BLANK_DELAY = 130000000; //2 seconds on 65mhz clock

input reset, clk;
input Manual_Override; //controls whether running in manual or automatic mode
input m_move_up, m_move_down, m_move_right, m_move_left; //manual move signals
input [5:0] v_angle_h, v_angle_v, a_angle; //automatic move signals
//note that the angles here use 0 degrees as the positive y axis and have 10 degree increments
//i.e. 90 degrees is represented by 9, 270 by 27, etc.
input v_motion, a_motion; //validates angles
input cant_move_up, cant_move_down; //overrides signals and prevents movement
input priority; //toggles what takes priority (high) audio or (low) video

output vIAp, vIAn, vIBp, vIBn; //signals sent to the vertical motor
output hIAp, hIAn, hIBp, hIBn; //signals sent to the horizontal motor
output look_up, look_down, look_right, look_left; //foward reverse signals sent to motor control
output moving; //high when motor is moving, low otherwise
output v_finished_moving, h_finished_moving; //finished moving signals
output v_expired, h_expired;
output blank;

reg look_up, look_down, look_left, look_right; //foward and reverse signals sent to motor control

//create motor control modules that handle the transition of stepper states when given fwd/active signals
motorcontrol vertMotor(reset, clk, look_up, look_down, vIAp, vIAn, vIBp, vIBn);
motorcontrol horizMotor(reset, clk, look_left, look_right, hIAp, hIAn, hIBp, hIBn);

//generate a one hz signal clk used to time how long to keep the motor signals active
wire four_hz_enable;
clock_divider div(clk, reset, four_hz_enable);
defparam div.clk_cycle = 27’d16200000; //4 times per second on a 65Mhz clock

reg v_start = 0, h_start = 0;
reg v_can_start = 1, h_can_start = 1;
reg [9:0] h_interval = 2, v_interval = 2;
//create timers that start whenever automatic motion is detected and will expire after "interval" pulses of the one_hz signal
timer v_timer(clk, reset, v_interval[9:0], v_start, four_hz_enable, v_expired);
timer h_timer(clk, reset, h_interval[9:0], h_start, four_hz_enable, h_expired);

//prevent motion on reset or motor expiration
reg [32:0] blank_count = BLANK_DELAY;
always @ (posedge clk) begin
    if (h_expired || reset) blank_count <= BLANK_DELAY; //put v_expired back
    if vertical motion is readded
        else if (blank_count > 0) blank_count <= blank_count - 1;
        else blank_count <= 0;
    end
assign blank = (blank_count != 0);

always @ (posedge clk) begin
    // finished_moving <= 0; //assume we have not finished moving and verify assumption when timers expire
    v_start <= 0; //don't start counting unless told to do so
    h_start <= 0;
    if (v_expired) begin
        v_can_start <= 1; //when we finish moving then signal that we can start again
        //stop motion:
        look_up <= 0;
        look_down <= 0;
    end
    if (h_expired) begin
        h_can_start <= 1; //when we finish moving then signal that we can start again
        //stop motion:
        look_right <= 0;
        look_left <= 0;
    end
    if (reset) begin
        //stop motion:
        look_up <= 0;
        look_down <= 0;
        look_right <= 0;
        look_left <= 0;
        //we can start moving again:
        v_start <= 0;
        v_can_start <= 1;
        v_interval <= 2; //2 was chosen arbitrarily, could really be anything > 0
        h_start <= 0;
        h_can_start <= 1;
        h_interval <= 2;
    end
    else if (Manual Override) begin
        //Manual Motion
        //assume no buttons are being pressed until determined otherwise:
        look_up <= 0;
        look_down <= 0;
        look_right <= 0;
        look_left <= 0;
        v_interval <= 2;
        h_interval <= 2;
    end
    if (m_move_up) begin //user wants to move up
        if (~cant_move_up) begin //make sure motor can move up
            look_up <= 1;
        end
    end
end
else if (m_move_down) begin //user wants to move down
    if (~cant_move_down) begin //make sure motor can
        look_down <= 1;
    end
end

if (m_move_right) begin //user wants to move clockwise
    look_right <= 1;
end
else if (m_move_left) begin //user wants to move ccw
    look_left <= 1;
end
end //end if(manual_override)

//Automatic Motion
//note that the angles here use 0 degrees as the positive y axis and have
10 degree increments
else if (~blank) begin //if we're not in a blanking period (delay period
after moving)
    if (~priority) begin //video motion takes priority
        if (v_motion) begin  //video motion detected,
            move camera horizontally and vertically
                if (v_angle_v <= 18) begin
                    //hold look_up high
                    if (v_can_start) begin
                        look_up <= 1;
                        v_can_start <= 0;
                        v_interval <= v_angle_v *
                            MOTOR_SPEED_FACTOR;
                    end
                    v_start <= 1;
                end
            else begin //hold look_down high
                if (v_can_start) begin
                    look_down <= 1;
                    v_can_start <= 0;
                    //since the angle is given
                    v_interval <= (36 -
                        v_angle_v) * MOTOR_SPEED_FACTOR;
                    v_start <= 1;
                end
            end
        end //end if(v_motion)
        else if (a_motion) begin //audio motion detected and
            no video motion
if (a_angle <= 18) begin
    //hold look_right high
    if (h_can_start) begin
        look_right <= 1;
        h_can_start <= 0;
        h_interval <= a_angle
    end

    h_start <= 1;
end
else begin
    //hold look_left high
    if (h_can_start) begin
        look_left <= 1;
        h_can_start <= 0;
        h_interval <= (36 - a_angle) * MOTOR_SPEED_FACTOR;
    end

    h_start <= 1;
end

end //end if(a_motion)

/* else begin //if there is no new audio or video
motion
look_up;
:look_down;
:look_left;
: look_right;

v_start <= 0;
: h_start <= 0;
end*/
end //end if(~priority)
else begin //audio motion takes priority
else if (a_motion) begin
    if (a_angle <= 18) begin
        //hold look_right high
        if (h_can_start) begin
            look_right <= 1;
            h_can_start <= 0;
            h_interval <= a_angle
        end

        h_start <= 1;
    end
else begin
    //hold look_left high
    if (h_can_start) begin
        look_left <= 1;
        h_can_start <= 0;
        h_interval <= (36 - a_angle) * MOTOR_SPEED_FACTOR;
    end

    h_start <= 1;
end
end //end if(a_motion)
else if (v_motion) begin //video motion detected, move camera horizontally and vertically
    if (v_angle_v <= 18) begin
        //hold look_up high
        if (v_can_start) begin
            look_up <= 1;
            v_can_start <= 0;
            v_interval <= v_angle_v * MOTOR_SPEED_FACTOR;
        end

        v_start <= 1;
    end
end
else begin
if (v_can_start) begin
  look_down <= 1;
  v_can_start <= 0;
  v_interval <= (36 - v_angle_v) * MOTOR_SPEED_FACTOR;
  v_start <= 1;
end

if (v_angle_h <= 18) begin
  //hold look_right high
  if (h_can_start) begin
    look_right <= 1;
    h_can_start <= 0;
    h_interval <= v_angle_h * MOTOR_SPEED_FACTOR;
    h_start <= 1;
  end
end
else begin
  //hold look_left high
  if (h_can_start) begin
    look_left <= 1;
    h_can_start <= 0;
    h_interval <= (36 - v_angle_h) * MOTOR_SPEED_FACTOR;
    h_start <= 1;
  end

  //end if(v_motion)
  /*else begin //if there is no new audio or video
  look_up <= v_expired || ~v_can_start ? 0 :
  look_down <= v_expired || ~v_can_start ? 0 :
  look_left <= h_expired || ~h_can_start ? 0 :
  look_right <= h_expired || ~h_can_start ? 0 :
  v_start <= 0;
  h_start <= 0;
  end       */
end  //end priority

else begin//if(blank)
  //stop motion for a brief interval after we have just moved to
  allow
  //video and audio processing units to gather new data
  look_up <= 0;
  look_down <= 0;
  look_left <= 0;
  look_right <= 0;
end

end //always

assign moving = look_up || look_down || look_left || look_right;
endmodule
module motorcontrol(reset, clk, fwd, rev, IAp, IAn, IBp, IBn);

parameter CLK_SPEED = 65000000; //65MHz clock
input reset, clk, fwd, rev;
output IAp, IAn, IBp, IBn; //outputs to the stepper motor coils (IA+, IA-, IB+, IB-)

reg [2:0] state = 0; //FSM w/ 8 states corresponding to the 8 different half-steps the motor can take
parameter wait_state = 5; //the state that the motor will stay in when neither fwd nor rev is asserted

//create a 5Hz clock to drive the motor
reg clk2 = 0;
reg [26:0] clock_count;
always @(posedge clk) begin
    if (reset) begin
        clock_count <= 0;
        clk2 <= 0;
    end
    else if (clock_count == CLK_SPEED/10) begin //alternate clk2 every 10th of a second, i.e. 5Hz
        clock_count <= 0;
        clk2 <= ~clk2;
    end
    else clock_count <= clock_count + 1;
end

always @(posedge clk) begin
    if (reset) begin
        state <= 0;
        // wait_state <= 1;
    end
    else if (clk2) begin
        if (fwd) begin
            state <= state + 1;
        end
        else if (rev) begin
            state <= state - 1;
        end
        else
            //state <= state; //don't change state if no signal is given
            state <= wait_state;
    end
end

//half step states:
//State IA+ IA- IB+ IB-
//s0 On Off On Off
//s1 Off Off On Off
//s2 Off Off On On
//s3 Off Off On Off
//s4 On Off Off On
//s5 On Off Off Off
//s6 Off On On Off
//s7 Off On On On
assign IAp = (state == 0) || (state == 1) || (state == 2);
assign IAn = (state == 4) || (state == 5) || (state == 6);
assign IBp = (state == 0) || (state == 6) || (state == 7);
assign IBn = (state == 2) || (state == 3) || (state == 4);

// Full step states:
// State IA+ IA- IB+ IB-
// s0    On  Off On  Off
// s1    On  Off Off On
// s2    Off On  Off On
// s3    Off On  On  Off

/*
assign IAp = (state == 0) || (state == 1);
assign IAn = (state == 2) || (state == 3);
assign Ibp = (state == 0) || (state == 3);
assign Ibn = (state == 1) || (state == 2);
*/

endmodule
module test_angle_generator(clk, reset, angle);
  input clk, reset;
  output [5:0] angle;

  parameter CLK_SPEED = 65000000;
  parameter INTERVAL = 20; //amount of time before increasing angle
  parameter MAX_VALUE = 35; //the maximum angle can be
  parameter START_VALUE = 0;
  parameter INCREMENT = 1; //how much to increase the angle each interval

  reg [25:0] clk_count = 0;
  reg [5:0] second_count = 0;
  reg [5:0] angle = START_VALUE;

  always @ (posedge clk) begin
    if(reset) begin
      angle <= START_VALUE;
      clk_count <= 0;
      second_count <= 0;
    end
    else if (second_count == INTERVAL) begin
      if (angle + INCREMENT >= MAX_VALUE) angle <= 0;
      else angle <= angle + INCREMENT;
      second_count <= 0;
    end
    if (clk_count == CLK_SPEED) second_count <= second_count + 1;
    clk_count <= clk_count + 1;
  end
endmodule
module timer(clk, reset, value, start_timer, hz_enable, expired);
  input clk, reset, start_timer, hz_enable;
  input [9:0] value;
  output expired;

  reg [9:0] count = 0;
  reg active = 0;

  always @ (posedge clk) begin
    if (reset) begin
      count <= 0;
      active <= 0;
    end
    else if (count > value || expired) begin
      active <= 0;
      count <= 0;
    end
    else if (start_timer) begin
      count <= 0;
      active <= 1;
    end

    if (active && hz_enable) begin
      count <= count + 1;
    end

    assign expired = active && (count == value);
  end
endmodule
/*Robert Speaker 6.111
turns a 65 mhz clock into a one hz clock.
*/

module clock_divider(clk_65, reset, one_hz_enable);
  input clk_65, reset;
  output one_hz_enable;
  parameter clk_cycle = 27'd65000000;
  //parameter clk_cycle = 3; //used for testing only
  reg [26:0] count = 27'b000000000000000000000000000; //2^26 >> 65,000,000
  always @ (posedge clk_65)
    begin
      if (reset) begin
        count <= 0;
      end
      //resets the count after enable is sent, set to one because one cycle has
      else if(one_hz_enable) count <= 1;
      else begin //otherwise increment count and enable is low
        count <= count + 1;
      end
    end
  assign one_hz_enable = (count == clk_cycle);
endmodule
module ad7871_controller(reset, clock_65mhz, convst_bar1, convst_bar2,  
convst_bar3, convst_bar4,  
  cs_bar1, cs_bar2, cs_bar3, cs_bar4,  
  rd_bar1, rd_bar2, rd_bar3, rd_bar4, int_bar1, int_bar2, int_bar3,  
  int_bar4, ready, state, reset_timer1,  
  reset_timer2, expired2, expired3, reset_timer3, expired3, count1,  
  reset_possible);
input reset;  
input clock_65mhz;  
input int_bar1, int_bar2, int_bar3, int_bar4;  
input [9:0] count1;  
output convst_bar1, convst_bar2, convst_bar3, convst_bar4;  
output cs_bar1, cs_bar2, cs_bar3, cs_bar4;  
output rd_bar1, rd_bar2, rd_bar3, rd_bar4;  
output ready;  
output [2:0] state;  
output reset_timer1, reset_timer2, reset_timer3;  
output reset_possible;  
reg convst_bar1, convst_bar2, convst_bar3, convst_bar4;  
reg cs_bar1, cs_bar2, cs_bar3, cs_bar4;  
reg rd_bar1, rd_bar2, rd_bar3, rd_bar4;  
reg ready;  
reg [2:0] state;  
reg [2:0] highstate;  
reg reset_timer1;  
reg reset_timer2;  
reg reset_timer3;  
reg reset_possible = 0;  
always @(posedge clock_65mhz)
begin  
  if ((count1 >= 10) && (count1 <= 100)) reset_possible <= 1;  
  else reset_possible <= 0;  
  if (reset)
    begin
      reset_timer1 <= 0;
      reset_timer2 <= 0;
      reset_timer3 <= 0;
      state <= 0;
      highstate <= 0;
    end
  else begin  
    //if (reset_timer1) reset_timer1 <= 0;  
    //if (reset_timer2) reset_timer2 <= 0;  
    //if (state > highstate) highstate <= state;  
    case (state)
      3'b000: //start convert pulse, wait for 2 clock cycles
        begin
          reset_timer1 <= 0;
          reset_timer2 <= expired2;
          reset_timer3 <= expired2;
          if (expired2) state <= 3'b001;
        end
      3'b001: //end convert pulse, wait for ~int_bar
        begin
          //...
        end
      endcase
    end
  end
reset_timer1 <= 0;
reset_timer2 <= expired3;
reset_timer3 <= expired3;
if (expired3/*~int_bar*/) state <= 3'b010;
end

3'b010: //start rd,cs pulse, wait for 2 clock cycles
begin
reset_timer1 <= 0;
reset_timer2 <= expired2;
reset_timer3 <= 0;
if (expired2) state <= 3'b011;
end

3'b011: //DATA READY!, wait for 2 clock cycles
begin
reset_timer1 <= 0;
reset_timer2 <= expired2;
reset_timer3 <= 0;
if (expired2) state <= 3'b100;
end

3'b100: //END PULSE, wait until convcount 300
begin
reset_timer1 <= expired1;
reset_timer2 <= (expired1);
reset_timer3 <= expired1;
if (expired1) state <= 3'b000;
end

3'b101: //reset int_bar, reset
begin
reset_timer1 <= expired2;
reset_timer2 <= expired2;
reset_timer3 <= 0;
if (expired2) state <= 3'b000;
end

default:
begin
if (expired2) state <= 3'b000;
end
endcase
end

always @(state)
begin
case(state)
3'b000:
begin
convst_bar1 = 0;
 cs_bar1 = 1;
 rd_bar1 = 1;
convst_bar2 = 0;
 cs_bar2 = 1;
 rd_bar2 = 1;
convst_bar3 = 0;
 cs_bar3 = 1;
 rd_bar3 = 1;
convst_bar4 = 0;
 cs_bar4 = 1;
 rd_bar4 = 1;
ready = 0;
end
3'b001:
begin
convst_bar1 = 1;
cs_bar1 = 1;
rd_bar1 = 1;
convst_bar2 =1;
cs_bar2 = 1;
rd_bar2 = 1;
convst_bar3 =1;
cs_bar3 = 1;
rd_bar3 = 1;
convst_bar4 =1;
cs_bar4 = 1;
rd_bar4 = 1;
ready = 0;
end

3'b010:
begin
    convst_bar1 =1;
cs_bar1 = 0;
rd_bar1 = 0;
convst_bar2 =1;
cs_bar2 = 0;
rd_bar2 = 0;
convst_bar3 =1;
cs_bar3 = 0;
rd_bar3 = 0;
convst_bar4 =1;
cs_bar4 = 0;
rd_bar4 = 0;
ready = 0;
end

3'b011:
begin
    convst_bar1 =1;
cs_bar1 = 0;
rd_bar1 = 0;
convst_bar2 =1;
cs_bar2 = 0;
rd_bar2 = 0;
convst_bar3 =1;
cs_bar3 = 0;
rd_bar3 = 0;
convst_bar4 =1;
cs_bar4 = 0;
rd_bar4 = 0;
ready = 1;
end

3'b100:
begin
    convst_bar1 =1;
cs_bar1 = 1;
rd_bar1 = 1;
convst_bar2 =1;
cs_bar2 = 1;
rd_bar2 = 1;
convst_bar3 =1;
cs_bar3 = 1;
rd_bar3 = 1;
convst_bar4 =1;
cs_bar4 = 1;
rd_bar4 = 1;
ready = 0;
end

3'b101:
begin
    convst_bar1 =1;
cs_bar1 = 0;
rd_bar1 = 0;
convst_bar2 = 1;
cs_bar2 = 0;
rd_bar2 = 0;
convst_bar3 = 1;
cs_bar3 = 0;
rd_bar3 = 0;
convst_bar4 = 1;
cs_bar4 = 0;
rd_bar4 = 0;
ready = 0;
end

default:
begin
convst_bar1 = 0;

endcase
end

endmodule
module anglecalc (reset, clock_65mhz, posangle1, posangle2, dir1, dir2, trueangle, a_motion);

input reset;
input clock_65mhz;
input [3:0] posangle1;
input [3:0] posangle2;
input [1:0] dir1;
input [1:0] dir2;

output [5:0] trueangle;
output a_motion;

reg [5:0] trueangle;
reg [3:0] avgangle;
reg a_motion;

always @(posedge clock_65mhz)
begin
if (reset)
begin
  trueangle <= 0;
  avgangle <= 0;
  a_motion <= 0;
end
else
begin
  avgangle <= (posangle1 + posangle2) >> 1;
  if (dir1 == 1 && dir2 == 1)
  begin
    a_motion <= 1;
    case (avgangle)
    0: trueangle <= 27;
    1: trueangle <= 28;
    2: trueangle <= 29;
    3: trueangle <= 30;
    4: trueangle <= 31;
    5: trueangle <= 32;
    6: trueangle <= 33;
    7: trueangle <= 34;
    8: trueangle <= 35;
    9: trueangle <= 0;
    default: trueangle <= 0;
    endcase
  end
  if (dir1 == 1 && dir2 == 2)
  begin
    a_motion <= 1;
    case (avgangle)
    0: trueangle <= 27;
    1: trueangle <= 26;
    2: trueangle <= 25;
    3: trueangle <= 24;
    4: trueangle <= 23;
    5: trueangle <= 22;
    6: trueangle <= 21;
    7: trueangle <= 20;
    8: trueangle <= 19;
    9: trueangle <= 18;
    default: trueangle <= 18;
    endcase
  end
  if (dir1 == 2 && dir2 == 1)
  begin
    a_motion <= 1;
    case (avgangle)
    0: trueangle <= 9;
    1: trueangle <= 8;
    2: trueangle <= 7;
    3: trueangle <= 6;
    default: trueangle <= 0;
    endcase
  end
end
end
if (dir1 == 2 && dir2 == 2)
begin
  a_motion <= 1;
  case (avgangle)
  0: trueangle <= 9;
  1: trueangle <= 10;
  2: trueangle <= 11;
  3: trueangle <= 12;
  4: trueangle <= 13;
  5: trueangle <= 14;
  6: trueangle <= 15;
  7: trueangle <= 16;
  8: trueangle <= 17;
  9: trueangle <= 18;
  default: trueangle <= 18;
  endcase
end
end
module timer1(reset, reset_timer1, clock_65mhz, value1, expired1, count1);

input reset;
input reset_timer1;
input clock_65mhz;
input [9:0] value1;
output expired1;
output [9:0] count1;

reg [9:0] count1;

always @ (posedge clock_65mhz)
begin
    if (reset || reset_timer1 || count1 >= value1) count1 <= 0;
    else count1 <= count1 + 1;
end

assign expired1 = (count1 >= value1);
endmodule

module timer2(reset, reset_timer2, clock_65mhz, value2, expired2, count2,
state);

input reset;
input reset_timer2;
input clock_65mhz;
input [3:0] value2;
//input int_bar;
input [2:0] state;
output expired2;
output [3:0] count2;

reg [3:0] count2;

always @ (posedge clock_65mhz)
begin
    if (reset || reset_timer2 || count2 >= value2) count2 <= 0;
    /*|| (~int_bar && (state == 3'b001))*/
    else count2 <= count2 + 1;
end

assign expired2 = (count2 >= value2);
endmodule

module timer3(reset, reset_timer3, clock_65mhz, value3, expired3, count3);

input reset;
input reset_timer3;
input clock_65mhz;
input [9:0] value3;
output expired3;
output [9:0] count3;

reg [9:0] count3;

always @ (posedge clock_65mhz)
begin
    if (reset || reset_timer3 || count3 >= value3) count3 <= 0;
    else count3 <= count3 + 1;
end

assign expired3 = (count3 >= value3);
endmodule

module timer4(reset, reset_timer4, clock_65mhz, value4, expired4, count4);

    input reset;
    input reset_timer4;
    input clock_65mhz;
    input [10:0] value4;
    output expired4;
    output [10:0] count4;

    reg [10:0] count4;

    always @ (posedge clock_65mhz)
        begin
            if (reset || reset_timer4 || count4 >= value4) count4 <= 0;
            else count4 <= count4 + 1;
        end

    assign expired4 = (count4 >= value4);

endmodule
module differencer (blank, count, reset, clock_65mhz, expired1, db1, db2, dir12, delay, state);

input blank;
input reset;
input clock_65mhz;
input expired1;
input [13:0] db1;
input [13:0] db2;

output [1:0] dir12;
output [9:0] delay;
output [2:0] state;
output [9:0] count;

reg [1:0] dir12;
reg [9:0] delay;
reg [2:0] state;
reg [9:0] count;

reg [13:0] threshold1;
reg [13:0] threshold2;

always @ (posedge clock_65mhz)
begin
    if (reset)
    begin
        dir12 <= 0;
        delay <= 10'b0000000000;
        state <= 3'b000;
        count <= 0;
        //threshold <= 13'h1554;
        //threshold1 <= 13'h1554;
        //threshold2 <= 13'h1554;
        threshold1 <= 14'h1554;
        threshold2 <= 14'h1554;
    end
    else if (~blank)
    begin
        case (state)
        3'b000:
            begin
                if (db1 >= threshold1)
                begin
                    state <= 3'b001;
                    count <= 0;
                    dir12 <= 1;
                end
            else if (db2 >= threshold2)
            begin
                state <= 3'b001;
                count <= 0;
                dir12 <= 2;
            end
            else
            begin
                //dir12 <= 0;
            end
        end
        3'b001:
    end
endmodule
begin
  if (expired1)
    begin
      if (db2 >= threshold2)
        begin
          delay <= count;
          count <= 0;
          dir12 <= 1;
          state <= 3'b011;
        end
      else if (count >= 10'b1111111111)
        begin
          dir12 <= 0;
          //count <= 0;
          state <= 3'b000;
        end
      else
        begin
          count <= count + 1;
          dir12 <= 1;
        end
    end
  end
3'b0010:
begin
  if (expired1)
    begin
      if (db1 >= threshold1)
        begin
          delay <= count;
          count <= 0;
          dir12 <= 2;
          state <= 3'b011;
        end
      else if (count >= 10'b1111111111)
        begin
          dir12 <= 0;
          //count <= 0;
          state <= 3'b000;
        end
      else
        begin
          count <= count + 1;
          dir12 <= 2;
        end
    end
  end
3'b011:
begin
/*if (expired1)
begin
  if (count >= 10'b1111111111)
    begin
      count <= 0;
      state <= 3'b011;
    end
  count <= count + 1;
end */
end
default:
begin
  dir12 <= 1;
  delay <= 0;
  count <= 0;
  state <= 0;
end
endcase
end
end
endmodule
module divider_big (
    dividend,
    divisor,
    quot,
    remd,
    clk,
    rfd,
    aclr,
    sclr,
    ce);    // synthesis black_box

input [13 : 0] dividend;
input [9 : 0] divisor;
output [13 : 0] quot;
output [9 : 0] remd;
input clk;
output rfd;
input aclr;
input sclr;
input ce;

// synopsys translate_off

SDIVIDER_V3_0 #(
    0, // c_has_aclr
    0, // c_has_ce
    0, // c_has_sclr
    1, // c_sync_enable
    1, // divclk_sel
    14, // dividend_width
    10, // divisor_width
    0, // fractional_b
    10, // fractional_width
    0) // signed_b

// synopsys translate_on
inst 
  .DIVIDEND(dividend),
  .DIVISOR(divisor),
  .QUOT(quot),
  .REMD(remd),
  .CLK(clk),
  .RFD(rfd),
  .ACLR(aclr),
  .SCLR(sclr),
  .CE(ce));

// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of divider_big is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of divider_big is "black_box"

endmodule
module divider_small (dividend, divisor, quot, remd, clk, rfd, aclr, sclr, ce);    // synthesis black_box
input [9 : 0] dividend;
input [9 : 0] divisor;
output [9 : 0] quot;
output [9 : 0] remd;
input clk;
output rfd;
input aclr;
input sclr;
input ce;

// synopsys translate_off
SDIVIDER_V3_0 #(
    0, // c_has_aclr
    0, // c_has_ce
    0, // c_has_sclr
    1, // c_sync_enable
    1, // divclk_sel
    10, // dividend_width
    10, // divisor_width
    0, // fractional_b
    10, // fractional_width
    0) // signed_b
// synopsys translate_on
}
```verilog
inst (  .DIVIDEND(dividend),  .DIVISOR(divisor),  .QUOT(quot),  .REMD(remd),  .CLK(clk),  .RFD(rfd),  .ACLR(aclr),  .SCLR(sclr),  .CE(ce));

// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of divider_small is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of divider_small is "black_box"

endmodule
```
module slopefinder (blank, reset, clock_65mhz, delay, dir12, x0, angle_out1, angle_out2, dividend, divisor, d_over_c, posangle);

input reset;
input clock_65mhz;
input [9:0] delay;
input [1:0] dir12;
input blank;

output [5:0] angle_out1;
output [5:0] angle_out2;
output [13:0] dividend;
output [9:0] divisor;
output [7:0] d_over_c;
output [7:0] x0;
output [3:0] posangle;

reg [5:0] angle_out1;
reg [5:0] angle_out2;
reg [13:0] dividend;
reg [9:0] divisor;
reg [7:0] x0;
wire [13:0] quot;
wire [9:0] remd;
reg [3:0] posangle;

wire [7:0] d_over_c;
assign d_over_c = quot[7:0];

divider_big div1(dividend, divisor, quot, remd, clock_65mhz, rfd);

always @ (posedge clock_65mhz)
begin
  if (reset)
  begin
    angle_out1 <= 0;
    angle_out2 <= 0;
    dividend <= delay;
    divisor <= 12;
    x0 <= 128;
  end
  else
  begin
    dividend <= delay*10;
    divisor <= 12;
    if (dir12 == 1)
    begin
      x0 <= 128-delay;
      if (d_over_c >= 0 && d_over_c <= 8)
      begin
        posangle <= 9;
        angle_out1 <= 0;
        angle_out2 <= 18;
      end
      else if (d_over_c >= 9 && d_over_c <= 25)
      begin
        posangle <= 8;
        angle_out1 <= 35;
        angle_out2 <= 19;
      end
      else if (d_over_c >= 26 && d_over_c <= 42)
      begin
        posangle <= 7;
        angle_out1 <= 52;
        angle_out2 <= 18;
      end
      else
      begin
        posangle <= 6;
        angle_out1 <= 69;
        angle_out2 <= 17;
      end
      end
  end
end
posangle <= 7;
angle_out1 <= 34;
angle_out2 <= 20;
end
else if (d_over_c >= 43 && d_over_c <= 57)
begin
posangle <= 6;
angle_out1 <= 33;
angle_out2 <= 21;
end
else if (d_over_c >= 58 && d_over_c <= 70)
begin
posangle <= 5;
angle_out1 <= 32;
angle_out2 <= 22;
end
else if (d_over_c >= 71 && d_over_c <= 81)
begin
posangle <= 4;
angle_out1 <= 31;
angle_out2 <= 23;
end
else if (d_over_c >= 82 && d_over_c <= 90)
begin
posangle <= 3;
angle_out1 <= 30;
angle_out2 <= 24;
end
else if (d_over_c >= 91 && d_over_c <= 96)
begin
posangle <= 2;
angle_out1 <= 29;
angle_out2 <= 25;
end
else if (d_over_c >= 97 && d_over_c <= 99)
begin
posangle <= 1;
angle_out1 <= 28;
angle_out2 <= 26;
end
else
begin
posangle <= 0;
angle_out1 <= 27;
angle_out2 <= 27;
end
end
else if (dir12 == 2)
begin
x0 <= 128+delay;
if (d_over_c >= 0 && d_over_c <= 8)
begin
posangle <= 9;
angle_out1 <= 0;
angle_out2 <= 18;
end
else if (d_over_c >= 9 && d_over_c <= 25)
begin
posangle <= 8;
angle_out1 <= 1;
angle_out2 <= 17;
end
else if (d_over_c >= 26 && d_over_c <= 42)
begin
posangle <= 7;
angle_out1 <= 2;
angle_out2 <= 16;
end
else if (d_over_c >= 43 && d_over_c <= 57)
begin
  posangle <= 6;
  angle_out1 <= 3;
  angle_out2 <= 15;
end
else if (d_over_c >= 58 && d_over_c <= 70)
begin
  posangle <= 5;
  angle_out1 <= 4;
  angle_out2 <= 14;
end
else if (d_over_c >= 71 && d_over_c <= 81)
begin
  posangle <= 4;
  angle_out1 <= 5;
  angle_out2 <= 13;
end
else if (d_over_c >= 82 && d_over_c <= 90)
begin
  posangle <= 3;
  angle_out1 <= 6;
  angle_out2 <= 12;
end
else if (d_over_c >= 91 && d_over_c <= 96)
begin
  posangle <= 2;
  angle_out1 <= 7;
  angle_out2 <= 11;
end
else if (d_over_c >= 97 && d_over_c <= 99)
begin
  posangle <= 8;
  angle_out1 <= 10;
  angle_out2 <= 26;
end
else
begin
  posangle <= 0;
  angle_out1 <= 9;
  angle_out2 <= 9;
end
end
endmodule

module locator (blank, reset, clock_65mhz, delay, dir12, x0, angle_out1,
angle_out2, dividend, divisor, d_over_c);
  input reset;
  input clock_65mhz;
  input [9:0] delay;
  input [1:0] dir12;
  input blank;

  output [5:0] angle_out1;
  output [5:0] angle_out2;
  output [9:0] dividend;
  output [9:0] divisor;
  output [7:0] d_over_c;
  output [7:0] x0;
reg [5:0] angle_out1;
reg [5:0] angle_out2;
reg [9:0] dividend;
reg [9:0] divisor;
reg [7:0] x0;
wire [9:0] quot;
wire [9:0] remd;
wire [7:0] d_over_c;
assign d_over_c = quot[7:0];
divider_small div1(dividend, divisor, quot, remd, clock_65mhz, rfd);

always @ (posedge clock_65mhz)
begin
  if (reset)
  begin
    angle_out1 <= 0;
    angle_out2 <= 0;
    dividend <= delay;
    divisor <= 12;
    x0 <= 128;
  end
  else
  begin
    dividend <= delay;
    divisor <= 12;
    if (dir12 == 1)
    begin
      x0 <= 128-delay;
      case (d_over_c)
        0:
        begin
          angle_out1 <= 0;
          angle_out2 <= 18;
        end
        1:
        begin
          angle_out1 <= 0;
          angle_out2 <= 18;
        end
        2:
        begin
          angle_out1 <= 35;
          angle_out2 <= 19;
        end
        3:
        begin
          angle_out1 <= 34;
          angle_out2 <= 20;
        end
        4:
        begin
          angle_out1 <= 34;
          angle_out2 <= 20;
        end
        5:
        begin
          angle_out1 <= 33;
          angle_out2 <= 21;
        end
        6:
        begin
          angle_out1 <= 32;
          angle_out2 <= 22;
        end
  end
end
begin
  angle_out1 <= 32;
  angle_out2 <= 22;
end
8:
begin
  angle_out1 <= 31;
  angle_out2 <= 23;
end
9:
begin
  angle_out1 <= 30;
  angle_out2 <= 24;
end
default:
begin
  angle_out1 <= 27;
  angle_out2 <= 27;
end
endcase
deright}

else if (dir12 == 2)
beg
begin
  x0 <= 128+delay;
  case (d_over_c)
  0:
    begin
      angle_out1 <= 0;
      angle_out2 <= 18;
    end
  1:
    begin
      angle_out1 <= 0;
      angle_out2 <= 18;
    end
  2:
    begin
      angle_out1 <= 1;
      angle_out2 <= 17;
    end
  3:
    begin
      angle_out1 <= 2;
      angle_out2 <= 16;
    end
  4:
    begin
      angle_out1 <= 2;
      angle_out2 <= 16;
    end
  5:
    begin
      angle_out1 <= 3;
      angle_out2 <= 15;
    end
  6:
    begin
      angle_out1 <= 4;
      angle_out2 <= 14;
    end
  7:
    begin
      angle_out1 <= 4;
      angle_out2 <= 14;
    end
  endcase
8:
begin
  angle_out1 <= 5;
  angle_out2 <= 13;
end
9:
begin
  angle_out1 <= 6;
  angle_out2 <= 12;
end

default:
begin
  angle_out1 <= 9;
  angle_out2 <= 9;
end
endcase
end
endmodule
module newlocator (reset, clock_65mhz, delay1, delay2, dir1, dir2, slope1, slope2, xpos, ypos, finalangle);
  input reset;
  input clock_65mhz;
  input [9:0] delay1;
  input [9:0] delay2;
  input [1:0] dir1;
  input [1:0] dir2;
  output [9:0] slope1;
  output [9:0] slope2;
  output [9:0] xpos;
  output [9:0] ypos;
  output [5:0] finalangle;

  wire [9:0] quot1;
  wire [9:0] remd1; //useless
  wire [9:0] quot2;
  wire [9:0] remd2; //useless
  reg [9:0] slope1;
  reg [9:0] slope2;

  wire [7:0] d_over_c1;
  wire [7:0] d_over_c2;

  assign d_over_c1 = quot1[7:0];
  assign d_over_c2 = quot2[7:0];

  divider_small div1(delay1, 10'd12, quot1, remd1, clock_65mhz, rfd1);
  divider_small div2(delay2, 10'd12, quot2, remd2, clock_65mhz, rfd2);

  wire [7:0] t2;
  wire [9:0] quot3;
  wire [9:0] remd3; //useless
  reg [9:0] t_dividend;
  reg [9:0] t_divisor;
  reg signed [9:0] xpos_shifted;
  reg signed [9:0] ypos_shifted;
  wire [9:0] xpos;
  wire [9:0] ypos;
  assign xpos = xpos_shifted-128;
  assign ypos = ypos_shifted;
  assign t2 = quot3[7:0];
  assign ypos = ypos_shifted;
  assign t2 = quot3[7:0];
  reg [13:0] invtan_dividend;
  reg [9:0] invtan_divisor;
  wire [13:0] quot4;
  wire [10:0] remd4;
  wire rfd;
  //assign finalslope = quot4[10:0];
  divider_small divt(t_dividend, t_divisor, quot3, remd3, clock_65mhz, rfd3);
  divider_big divinvatan(invtan_dividend, invtan_divisor, quot4, remd4, clock_65mhz, rfd4);

  reg [5:0] finalangle;

  always @ (posedge clock_65mhz)
  begin
    if (reset)
begin
    slope1 <= 0;
slope2 <= 0;
t_dividend <= 0;
t_divisor <= 1;
xpos_shifted <= 0;
ypos_shifted <= 0;
invtan_dividend <= 0;
invtan_divisor <= 1;
finalangle <= 0;
end
else
begin
    case (d_over_c)
    0:
    begin
        slope1 <= 100;
    end
    1:
    begin
        slope1 <= 10;
    end
    2:
    begin
        slope1 <= 5;
    end
    3:
    begin
        slope1 <= 3 + 1/4;
    end
    4:
    begin
        slope1 <= 2 + 1/4;
    end
    5:
    begin
        slope1 <= 1 + 3/4;
    end
    6:
    begin
        slope1 <= 1 + 1/4;
    end
    7:
    begin
        slope1 <= 1;
    end
    8:
    begin
        slope1 <= 3/4;
    end
    9:
    begin
        slope1 <= 1/2;
    end
    default:
    begin
        slope1 <= 0;
    end
endcase

case (d_over_c2)
0:
begin
    slope2 <= 100;
end
1:
begin
  slope2 <= 10;
end
2:
begin
  slope2 <= 5;
end
3:
begin
  slope2 <= 3 + 1/4;
end
4:
begin
  slope2 <= 2 + 1/4;
end
5:
begin
  slope2 <= 1 + 3/4;
end
6:
begin
  slope2 <= 1 + 1/4;
end
7:
begin
  slope2 <= 1;
end
8:
begin
  slope2 <= 3/4;
end
9:
begin
  slope2 <= 1/2;
end
default:
begin
  slope2 <= 0;
end
endcase

if (dir1 == 1 && dir2 == 1)
begin
  t_dividend <= slope1*delay1+delay2;
  t_divisor <= 2*slope1*slope2-2;
  xpos_shifted <= 60-slope2*t2;
  ypos_shifted <= delay2/2+t2;
  invtan_dividend <= -xpos;
  invtan_divisor <= ypos*10;
  if (finalslope == 0 || finalslope == 1) finalangle <= 27;
  else if (finalslope == 2) finalangle <= 28;
  else if (finalslope == 3 || finalslope == 4) finalangle <= 29;
  else if (finalslope >= 5 || finalslope == 7) finalangle <= 30;
  else if (finalslope == 8 || finalslope == 9) finalangle <= 31;
  else if (finalslope >= 10 || finalslope <= 14) finalangle <= 32;
  else if (finalslope >= 15 || finalslope <= 21) finalangle <= 33;
  else if (finalslope >= 22 || finalslope <= 37) finalangle <= 34;
  else if (finalslope >= 38 || finalslope <= 114) finalangle <= 35;
  else finalangle <= 0;
end
else if (dir1 == 2 || dir2 == 1)
begin
  t_dividend <= slope1*delay1+delay2;
  t_divisor <= 2*slope1*slope2-2;
  xpos_shifted <= 60+slope2*t2;
  ypos_shifted <= delay2/2+t2;
  invtan_dividend <= -xpos;
  invtan_divisor <= ypos*10;
  if (finalslope == 0 || finalslope == 1) finalangle <= 27;
  else if (finalslope == 2) finalangle <= 28;
  else if (finalslope == 3 || finalslope == 4) finalangle <= 29;
  else if (finalslope >= 5 || finalslope == 7) finalangle <= 30;
  else if (finalslope == 8 || finalslope == 9) finalangle <= 31;
  else if (finalslope >= 10 || finalslope <= 14) finalangle <= 32;
  else if (finalslope >= 15 || finalslope <= 21) finalangle <= 33;
  else if (finalslope >= 22 || finalslope <= 37) finalangle <= 34;
  else if (finalslope >= 38 || finalslope <= 114) finalangle <= 35;
  else finalangle <= 0;
end
invtan_divisor <= ypos*10;
if (finalslope == 0 || finalslope == 1) finalangle <= 9;
else if (finalslope == 2) finalangle <= 8;
else if (finalslope == 3 || finalslope == 4) finalangle <= 7;
else if (finalslope >= 5 || finalslope <= 7) finalangle <= 6;
else if (finalslope == 8 || finalslope == 9) finalangle <= 5;
else if (finalslope >= 10 || finalslope <= 14) finalangle <= 4;
else if (finalslope >= 15 || finalslope <= 21) finalangle <= 3;
else if (finalslope >= 22 || finalslope <= 37) finalangle <= 2;
else if (finalslope >= 38 || finalslope <= 114) finalangle <= 1;
else finalangle <= 0;
end

else if (dir1 == 1 && dir2 == 2)
begin
  t_dividend <= slope1*delay1-240-delay2;
  t_divisor <= 2*slope1*slope2-2;
  xpos_shifted <= 60-slope2*t2;
  ypos_shifted <= 120+delay2/2-t2;
  invtan_dividend <= -xpos;
  invtan_divisor <= -ypos*10;
  if (finalslope == 0 || finalslope == 1) finalangle <= 27;
  else if (finalslope == 2) finalangle <= 26;
  else if (finalslope == 3 || finalslope == 4) finalangle <= 25;
  else if (finalslope >= 5 || finalslope <= 7) finalangle <= 24;
  else if (finalslope == 8 || finalslope == 9) finalangle <= 23;
  else if (finalslope >= 10 || finalslope <= 14) finalangle <= 22;
  else if (finalslope >= 15 || finalslope <= 21) finalangle <= 21;
  else if (finalslope >= 22 || finalslope <= 37) finalangle <= 20;
  else if (finalslope >= 38 || finalslope <= 114) finalangle <= 19;
  else finalangle <= 18;
end

else if (dir1 == 2 && dir2 == 2)
begin
  t_dividend <= slope1*delay1-240-delay2;
  t_divisor <= 2*slope1*slope2-2;
  xpos_shifted <= 60+slope2*t2;
  ypos_shifted <= 120+delay2/2-t2;
  invtan_dividend <= xpos;
  invtan_divisor <= -ypos*10;
  if (finalslope == 0 || finalslope == 1) finalangle <= 9;
  else if (finalslope == 2) finalangle <= 10;
  else if (finalslope == 3 || finalslope == 4) finalangle <= 11;
  else if (finalslope >= 5 || finalslope <= 7) finalangle <= 12;
  else if (finalslope == 8 || finalslope == 9) finalangle <= 13;
  else if (finalslope >= 10 || finalslope <= 14) finalangle <= 14;
  else if (finalslope >= 15 || finalslope <= 21) finalangle <= 15;
  else if (finalslope >= 22 || finalslope <= 37) finalangle <= 16;
  else if (finalslope >= 38 || finalslope <= 114) finalangle <= 17;
  else finalangle <= 18;
end
end
endmodule
//ray wu: module calc_camera_angle
//passes v and h angle to mechanical system to move camera
//last updated 12-11-05
module calc_camera_angle(reset, clock, calc_angle, mb_row, mb_col, v_angle_h, v_angle_v, angle_calculated);
module calc_camera_angle(reset, clock, calc_angle, motion_center, v_angle_h, v_angle_v, angle_calculated, debug);

input reset;
input clock;
input calc_angle;       //triggers the calculation
input [8:0] motion_center;     //motion center macroblock
output [5:0] v_angle_h;
output [5:0] v_angle_v;
output angle_calculated; //ok to move camera
output [63:0] debug;
reg [5:0] v_angle_h;
reg [5:0] v_angle_v;
reg angle_calculated;

//movement history: if the motion center is always at the same place..probably
noise..don't send signal for movement
reg [8:0] history_center_mb;

//center mb is at row=6, col=16
//assumptions: camera moves at 10 degree increments. Object is apprx. 1 meter away
from camera.
//      row movement: for every 10 degrees, object is apprx. 9 mb's away from
mb_row 16.
always @(posedge clock)
begin
if (reset)
begin
v_angle_h <= 0;
v_angle_v <= 0;
angle_calculated <= 0;
end
else if (calc_angle)
begin
if ( (motion_center[4:0] != history_center_mb[4:0]) & //noise
threshold row
(motion_center[8:5] != history_center_mb[8:5]) )  //noise
threshold col
begin
angle_calculated <= 1;
//calculate row difference
if (motion_center[8:5] > 8) //leave a margin of error->don't move if
it's within +- 2 mb's
v_angle_v <= 6'd34; //move 10 degrees down
else if (motion_center[8:5] < 4)
v_angle_v <= 6'd2; //move 10 degrees up
else
v_angle_v <= 6'd0;
//calculate col difference
if (motion_center[4:0] > 24) //leave a margin of error->don't move if it's
within +- 2 mb's
v_angle_h <= 6'd4; //move 10 degrees to the right
else if (motion_center[4:0] > 17)
v_angle_h <= 6'd2; //move 20 degrees to the right
else if (motion_center[4:0] < 8) //leave a margin of error->don't move
if it's within +- 2 mb's
v_angle_h <= 6'd32; //move 20 degrees to the right
else if (motion_center[4:0] < 15)
v_angle_h <= 6'd34; //move 10 degrees to the right
else
v_angle_h <= 6'd0;
end
else
v_angle_h <= 6'd0;
end
begin
  angle_calculated <= 0;
end

history_center_mb <= motion_center;

else
  begin
    angle_calculated <= 0;
  end
end

//debug outputs
assign debug = {48'b0, 2'b0, v_angle_v, 2'b0, v_angle_h};

endmodule
module calc_mberror(reset, clock, recording, done_retrieving_data, difference, difference_ready, mb_num, motion_center, center_ready, debug, motion_mb1, motion_mb2, motion_mb3, motion_mb4);

input reset;
input clock;            //v_clock
input recording;    //when ntsc_to_ram is recording a frame
input done_retrieving_data; //when high, means video memory retrieval is done
input [7:0] difference;     //luma difference for a pixel
input difference_ready;    //can process
input [8:0] mb_num;        //384 mb's total...0 to 383
output [8:0] motion_center; //center of the motion (row(4), col(5))
output center_ready;        //when center macroblock has been determined
reg center_ready = 0;
output [63:0] debug;
output [8:0] motion_mb1;
output [8:0] motion_mb2;
output [8:0] motion_mb3;
output [8:0] motion_mb4;

reg [13:0] mb_sad_tot_temp;
reg [2:0] lowest_sad_mb; //chooses between one of the mb_high_sad's
reg [13:0] lowest_sad;
reg [13:0] mb_high_sad[3:0]; //store 4 highest mb's
reg [8:0] mb_high_index[3:0];
reg [6:0] mb_pixel_counter = 0; //count to 64 (used to be [6:0])
//rely on mb_num to determine current MB
reg [5:0] row_total;
reg [6:0] col_total;
wire [3:0] average_crow = row_total[5:2]; //divide by 4
wire [4:0] average_ccol = col_total[6:2]; //divide by 4
assign motion_center = {average_crow, average_ccol};

always @(posedge clock)
begin
   if (reset | recording)
   begin
     //initialize state variables
     mb_pixel_counter <= 7'd0; //counter to 64 (8x8 pixels per macroblock)
     center_ready <= 1'd0;
     //initialize data variables
     lowest_sad_mb <= 3'd0;
     lowest_sad <= 14'd0;
     mb_sad_tot_temp <= 14'd0;
     mb_high_sad[0] <= 14'd0;
     mb_high_sad[1] <= 14'd0;
     mb_high_sad[2] <= 14'd0;
     mb_high_sad[3] <= 14'd0;
     mb_high_index[0] <= 9'd0;
     mb_high_index[1] <= 9'd1;
     mb_high_index[2] <= 9'd2;
     mb_high_index[3] <= 9'd3;
     row_total <= 6'd0;
     col_total <= 7'd0;
   end
end //video processing time
if (done_retrieving_data) //we've finished processing a frame
begin
  if ((mb_high_sad[0] + mb_high_sad[1] + mb_high_sad[2] +
      mb_high_sad[3]) > THRESHOLD_SAD)
  begin
    //past noise threshold, we have real motion
    center_ready <= 1;
    col_total <= mb_high_index[0][4:0] + mb_high_index[1][4:0]
    + mb_high_index[2][4:0] + mb_high_index[3][4:0];
    row_total <= mb_high_index[0][8:5] + mb_high_index[0][8:5]
    + mb_high_index[2][8:5];
  end
  else //not finished retrieving all luma data
  begin
    center_ready <= 0;
  end
end

else //not finished retrieving all luma data
begin
  if (difference_ready) //if calc_luma_difference calculates a
calculate the lowest mb, lowest sad
begin
if (mb_num == 9'd0)
  mb_high_sad[0] <= mb_high_sad[0] + difference;
else if (mb_num == 9'd1)
  mb_high_sad[1] <= mb_high_sad[1] + difference;
else if (mb_num == 9'd2)
else if (mb_num == 9'd3)
else if ((mb_num == 9'd4) & (mb_pixel_counter == 0))
begin
  //on first pass after first 4 mb's are entered,
  //lowest sad may eventually be replaced
  if ((mb_high_sad[0] < mb_high_sad[1]) & (mb_high_sad[0] <
      mb_high_sad[2]))
  begin
    lowest_sad mb <= 0; //mb index 0 has lowest sad
    lowest_sad <= mb_high_sad[0];
  end
else if ((mb_high_sad[1] < mb_high_sad[0]) & (mb_high_sad[1] <
      mb_high_sad[2]))
  begin
    lowest_sad mb <= 1; //mb index 1 has lowest sad
    lowest_sad <= mb_high_sad[1];
  end
else if ((mb_high_sad[2] < mb_high_sad[0]) & (mb_high_sad[2] <
      mb_high_sad[1]))
  begin
    lowest_sad mb <= 2; //mb index 2 has lowest sad
    lowest_sad <= mb_high_sad[2];
  end
else
begin
    lowest_sad mb <= 3; //mb index 3 has lowest sad
    lowest_sad <= mb_high_sad[3];
end
mb_sad_tot_temp <= difference;
mb_pixel_counter <= 1;
end
else //after initialization of 4 mb_high_sad (mb_num >=
4)
begin

if (mb_pixel_counter == 63) //right after we
begin
    //check if newest calculated SAD is bigger
    if ( ((mb_sad_tot_temp + difference) >
            ((mb_num[4:0])*32 < 992) ) //don't
    begin
        //replace the lowest SAD
        mb_high_index[lowest_sad_mb] <=
            mb_high_sad[lowest_sad_mb] <=
            //find the new lowest_sad index and
            if ( (mb_sad_tot_temp + difference) <
                (mb_sad_tot_temp + difference) <
                (mb_sad_tot_temp + difference) <=
                (mb_sad_tot_temp + difference) <
            begin
                //the lowest mb replaced is still
                lowest_sad_mb <= lowest_sad_mb;
                lowest_sad <=
            end
        else if ( (mb_high_sad[(lowest_sad_mb+1)%4] <
                    (mb_sad_tot_temp + difference) &
                    (mb_high_sad[(lowest_sad_mb+1)%4] <
                    mb_high_sad[(lowest_sad_mb+2)%4]) &
                    (mb_high_sad[(lowest_sad_mb+3)%4])
            begin
                //lowest mb sad is the next mb
                lowest_sad_mb <= (lowest_sad_mb+1)%4;
                lowest_sad <=
            end
        end
    else if ( (mb_high_sad[(lowest_sad_mb+2)%4]
                < mb_high_sad[(lowest_sad_mb+1)%4]) &
                < mb_high_sad[(lowest_sad_mb+3)%4])
            begin
                //lowest mb sad is the 2 down mb
                lowest_sad_mb <= (lowest_sad_mb+2)%4;
                lowest_sad <=
            end
        else
            begin
                //lowest mb sad is the 3 down mb
                lowest_sad_mb <= (lowest_sad_mb+3)%4;
                lowest_sad <=
            end
    end
else //we're not at the end of a macroblock yet
begin
    mb_sad_tot_temp <= mb_sad_tot_temp +
    difference; //add the difference to the macroblock
end
mb_pixel_counter <= mb_pixel_counter + 1;
end
end //mb != 0, 1, 2, 3
end //difference_ready
end //done_retrieving data
end //reset
end //always

assign motion_mb1 = mb_high_index[0];
assign motion_mb2 = mb_high_index[1];
assign motion_mb3 = mb_high_index[2];
assign motion_mb4 = mb_high_index[3];

//debug outputs
assign debug = {2'd0, mb_high_sad[3], 2'd0, mb_high_sad[2], 2'd0, mb_high_sad[1],
2'd0, mb_high_sad[0]};
endmodule
module vga_with_ram (reset, clock_27mhz, clock_65mhz, 
    vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b, vga_out_blank_b, 
    vga_out_pixel_clock, vga_out_hsync, vga_out_vsync, vram_addr, vram_data_in, 
    vram_data_out, vram_clk, vram_we, 
    recording, center_ready, motion_center, v_angle_h, v_angle_v, angle_calculated, debug, mb_output); 
//ray wu: vga is for testing purposes, but add code in this module to calculate angle from stored memory 
//last updated 11-28-05

ingut reset; // Active high reset, synchronous with 27MHz clock 
ingut clock_27mhz; //27MHz input clock 
//input v_proc_clock; //tv clock used in video processing (use the same clock) 
ingut clock_65mhz; 
output [7:0] vga_out_red, vga_out_green, vga_out_blue; // Outputs to DAC 
ingut vga_out_sync_b, vga_out_blank_b; // Composite sync/blank outputs to DAC 
ingut vga_out_pixel_clock; //Pixel clock for DAC 
ingut vga_out_hsync, vga_out_vsync; // Sync outputs to VGA connector 

input [15:0] vram_addr; //video ram address 
ingut [7:0] vram_data_in; //video ram data input 
ingut vram_clk, vram_we; //video ram clock and write enable 
ingut [7:0] vram_data_out; //video ram data output 

//ray wu 
ingut recording; //when ntsc_to_ram is recording a frame (use clock cycles for calculation when not record) 
output center_ready; //triggers calculation of angle 
output [8:0] motion_center; 
output [5:0] v_angle_h; //horizontal angle change 
output [5:0] v_angle_v; //vertical angle change 
output angle_calculated; //output from angle module ready 
output [63:0] debug; //to led/user3's for debugging 
output [8:0] mb_output; 

////////////////////////////////////////////////////////
/
// Timing values 
/
////////////////////////////////////////////////////////

// 1024 X 768 @ 75Hz with a 78.750MHz pixel clock

`define H_ACTIVE 1024 // pixels 
`define H_FRONT_PORCH 16 // pixels 
`define H_SYNCH 96 // pixels 
`define H_BACK_PORCH 176 // pixels 
`define H_TOTAL 1312 // pixels 

`define V_ACTIVE 768 // lines 
`define V_FRONT_PORCH 3 // lines 
`define V_SYNCH 9 // lines 
`define V_BACK_PORCH 28 // lines 
`define V_TOTAL 800 // lines */

//Changed by Bobby 12/12
//we only need a 65Mhz clock because the LCD's we are using don't care about the faster frame rate
// 1024 X 768 @ 60Hz with a 65MHz pixel clock
`define H_ACTIVE 1024 // pixels 
`define H_FRONT_PORCH 16 // pixels 
`define H_SYNCH 96 // pixels 
`define H_BACK_PORCH 160 // pixels 
`define H_TOTAL 1296 // pixels 

`define V_ACTIVE 768 // lines 
`define V_FRONT_PORCH 6 // lines 
`define V_SYNCH 6 // lines 
`define V_BACK_PORCH 29 // lines
\`define V_TOTAL 806 // lines

////////////////////////////////////////////////////////////////////////////
//
// Internal signals
//
////////////////////////////////////////////////////////////////////////////
wire pixel_clock;
reg prst, pixel_reset; // Active high reset, synchronous with pixel clock

reg [7:0] vga_out_red, vga_out_blue, vga_out_green;
wire vga_out_sync_b, vga_out_blank_b;
reg hsync1, hsync2, vga_out_hsync, vsync1, vsync2, vga_out_vsync;

reg [10:0] pixel_count; // Counts pixels in each line
reg [10:0] line_count; // Counts lines in each frame
reg [9:0] xpos; // horizontal image pixel count
reg [9:0] ypos; // vertical image pixel count

////////////////////////////////////////////////////////////////////////////
//
// Generate the pixel clock (78.750MHz)
//
////////////////////////////////////////////////////////////////////////////
// synthesis attribute period of clock_27mhz is 37ns;
/*DCM vga_dcm (.CLKIN(clock_27mhz),
 .RST(1'b0),
 .CLKFX(pixel_clock));
 // synthesis attribute DLL_FREQUENCY_MODE of vga_dcm is "LOW"
 // synthesis attribute DUTY_CYCLE_CORRECTION of vga_dcm is "TRUE"
 // synthesis attribute STARTUP_WAIT of vga_dcm is "TRUE"
 // synthesis attribute DFS_FREQUENCY_MODE of vga_dcm is "LOW"
 // synthesis attribute CLKFX_DIVIDE of vga_dcm is 9
 // synthesis attribute CLKFX_MULTIPLY of vga_dcm is 26
 // synthesis attribute CLK_FEEDBACK of vga_dcm is "NONE"
 // synthesis attribute CLKOUT_PHASE_SHIFT of vga_dcm is "NONE"
 // synthesis attribute clkin_period of vga_dcm is 37*/
assign pixel_clock = clock_65mhz;
assign vga_out_pixel_clock = ~pixel_clock;

always @(posedge pixel_clock)
begin
  prst <= reset;
  pixel_reset <= prst;
end

////////////////////////////////////////////////////////////////////////////
//
// Pixel and Line Counters
//
////////////////////////////////////////////////////////////////////////////
always @(posedge pixel_clock)
if (pixel_reset)
begin
  pixel_count <= 0;
  line_count <= 0;
end
else if (pixel_count == (`H_TOTAL-1)) // last pixel in the line
begin
  pixel_count <= 0;
  if (line_count == (`V_TOTAL-1)) // last line of the frame
  begin
    line_count <= 0;
  end
else
line_count <= line_count + 1;
else	pixel_count <= pixel_count + 1;

always @(pixel_reset or pixel_count or line_count)
if (pixel_reset)
begin
xpos <= 0;
ypos <= 0;
end
else begin
if (pixel_count > `H_FRONT_PORCH)
xpos <= pixel_count - `H_FRONT_PORCH;
if (line_count > `V_FRONT_PORCH)
ypos <= line_count - `V_FRONT_PORCH;
end

////////////////////////////////////////////////////////////////////////////
// Sync and Blank Signals
////////////////////////////////////////////////////////////////////////////

always @(posedge pixel_clock)
begin
if (pixel_reset)
begin
hsync1 <= 1;
hsync2 <= 1;
vga_out_hsync <= 1;
vsync1 <= 1;
vsync2 <= 1;
vga_out_vsync <= 1;
end
else
begin

// Horizontal sync
if (pixel_count == (`H_ACTIVE+`H_FRONT_PORCH))
hsync1 <= 1; // start of h_sync
else if (pixel_count == (`H_ACTIVE+`H_FRONT_PORCH+`H_SYNCH))
hsync1 <= 1; // end of h_sync

// Vertical sync
if (pixel_count == (`H_TOTAL+1))
begin
if (line_count == (`V_ACTIVE+`V_FRONT_PORCH))
vsync1 <= 0; // start of v_sync
else if (line_count == (`V_ACTIVE+`V_FRONT_PORCH+`V_SYNCH))
vsync1 <= 1; // end of v_sync
end

end

// Delay hsync and vsync by two cycles to compensate for 2 cycles of
// pipeline delay in the DAC.
hsync2 <= hsync1;
vga_out_hsync <= hsync2;
vsync2 <= vsync1;
vga_out_vsync <= vsync2;
end

// Blanking
assign vga_out_blank_b = ((pixel_count<`H_ACTIVE) & (line_count<`V_ACTIVE));

// Composite sync
assign vga_out_sync_b = hsync1 ^ vsync1;

////////////////////////////////////////////////////////////////////////////
Display a 256x192 pixel image from dual-port RAM

```verilog
reg [7:0] pixdata; // for the memory read pipeline
reg [15:0] memaddr; // for memory read address
wire [7:0] memdata; // for memory output data 8 bits: BBGGGRRR

// read data from memory at pixel clock, with one pipeline stage
always @(posedge pixel_clock)
begin
  memaddr <= xpos[9:2] + ypos[9:2]*256; // oversample
  pixdata <= memdata; // latch in last value
end
```

Ray Wu

New RAM can store 2 frames: frame A -> (cols=256, rows=96), frameB -> (256, 96) but row shifted down 96

Storage for luma values extracted from vram

```verilog
reg [7:0] lumaA = 0; //luma value from frameA
reg [7:0] lumaB = 0; //luma value from frameB

//used for the video block, but in same way as memaddr (change vram's memaddr to v_memaddr)
  reg [15:0] v_memaddr = 0;
```

Each macroblock is 8x8: total 384, 0 is upper left, 383 is lower right block

```verilog
reg [8:0] macroblock = 0; //macroblock number (goes left to right, up to down) (512 capacity)
reg [6:0] pixelsFed = 0; //count to 64 for each macroblock (128 capacity)
reg getLumaA = 1;
reg [7:0] difference_luma = 0;
reg difference_ready = 0;
reg done_retrieving_data;
wire [7:0] nextFrame_row_shift = {1'b0, macroblock[8:5], pixelsFed[5:3]} + 8'd96;
```

Note: we don't care in video processing about displaying the pixels (so timing in sync with vga is not needed)

```verilog
always @(posedge pixel_clock) //drive at65MHz
begin
  if (difference_ready) difference_ready <= 0; //reset the trigger if it's high
  if (reset | recording)
  begin
    //reset all variables
    macroblock <= 0;
pixelsFed <= 0;
    getLumaA <= 1; //start with retrieving info from frameA
    done_retrieving_data <= 0;
difference_ready <= 0;
difference_luma <= 0;
  end
  else
  begin
    //memaddr = {row,col}
    //procedure: first get luma from frame A, then from frame B -> calc difference of lumas, and out to calc_mberror
    // repeat for each mb (8x8px/mb, 32x12 mb's/frame
    if (!done_retrieving_data)
    begin
      //if (macroblock == 384) //if we finished processing every macroblock
```
begin
  done_retrieving_data <= 1; //turn off any more data collection
end
else //more macroblocks to process
begin
  if (pixelsFed == 64) //move to next macroblock
      begin
        getLumaA <= 1; //start new mb calc by getting luma from A
        pixelsFed <= 0;
        macroblock <= macroblock + 1;
      end
  else //get more pixels from frameA and frameB
      begin
        if (getLumaA) //get luma from frame A {row(4), col(5)}
            begin
                //note: memaddr (row, col) refers to pixels,
                //not macroblocks, so need to convert
                v_memaddr <= { 1'b0, macroblock[8:5],
                              macroblock[4:0],
                              pixelsFed[5:3],
                              pixelsFed[2:0] };
                lumaA <= memdata; //record in register
                getLumaA <= 1'b0; //now get luma from B
            end
        else //get luma from frame B {row(4), col(5)}
            begin //note: row shifted by 96 for memory of frame B
                v_memaddr <= { nextFrame_row_shift,
                              macroblock[4:0],
                              pixelsFed[2:0] };
                //lumaB <= memdata; just compute |lumaA - lumaB|...high for 1 cycle
                getLumaA <= 1'b1; //next step get luma from A
            end
        end //not pixelsFed = 64
      end //macroblock
end //done_retrieving_data
end //pixel_clock

//note: SAD is calculated in real time...per macroblock
//if difference is calculated...difference_ready triggers high...so calc_mberror takes
the difference into the SAD
//the above block outputs go_calc_luma, lumaA, and lumaB to the NEXT module (calc_mberror)

//send difference values to calculation of macroblock error
//calculate error per macroblock and output the motion center block

//test
wire [8:0] motion_mb1;
wire [8:0] motion_mb2;
wire [8:0] motion_mb3;
wire [8:0] motion_mb4;
wire [8:0] motion_center;
wire [63:0] debug_mberror;
calc_mberror calc_mberror1(.reset(reset), .clock(pixel_clock), .recording(recording),
    .done_retrieving_data(done_retrieving_data),
    .difference(difference_luma),
    .difference_ready(difference_ready), .mb_num(macroblock),
    .motion_center(motion_center),
    .center_ready(center_ready),
    .debug(debug_mberror), .motion_mb1(motion_mb1),
    .motion_mb2(motion_mb2), .motion_mb3(motion_mb3), .motion_mb4(motion_mb4));

//to bobby's master control module
wire [63:0] debug_camera_angle;
calc_camera_angle calc_camera_angle1(.reset(reset), .clock(pixel_clock),
    .calc_angle(center_ready),
    .motion_center(motion_center),
    .v_angle_v(v_angle_v),
    .v_angle_h(v_angle_h),
    .angle_calculated(angle_calculated),
    .debug(debug_camera_angle));

//when calc_mberror has a valid output (motion center macroblock)...center_ready goes high...
//data in center_mb_row, center_mb_col
reg [15:0] memaddr_select;
reg write_select;
always @(posedge pixel_clock)
begin
    if (reset)
        begin
            memaddr_select <= memaddr;
            write_select <= vram_we;
        end
    else
        begin
            //select the right memory address and write_enable
            if (recording | done_retrieving_data)
                begin
                    //go to default state
                    memaddr_select <= memaddr;
                    write_select <= vram_we; //go
                end
            else
                begin
                    memaddr_select <= v_memaddr;
                    write_select <= 0; //no writing to memory...only reading
                end
        end
end
//assign vram_addr_wire = memaddr_select;
//assign vram_write_wire = write_select;
videoram vram(.addra(memaddr_select), .addrb(vram_addr), .clka(pixel_clock),
    .clkb(vram_clk), .dinb(vram_data_in),
    .douta(memdata), .doutb(vram_data_out), .web(write_select));

// RAM with a 256 x 192 bw image, 8-bits per pixel
// the vga output is 1024x768, so we skip 2 bits of xpos and ypos, each
// feed pixel values to VGA machine
always @(posedge pixel_clock)
begin
    if (!recording) //if we're done computing (then next frame...draw the center motion block)
        begin
            //draw the macroblock
if (done_retrieving_data)
begin
    //if (center_ready & (xpos >=
        (xpos <
        (ypos >=
        (ypos <
        if (xpos >= 992)
            begin
                vga_out_red <= 128;  // weird color block
                vga_out_green <= 128;
                vga_out_blue <= 128;
            end
        else if (center_ready & (xpos >= 32*(motion_center[4:0])) & (xpos <
        (32*(motion_center[4:0])+32)) &
        (ypos >= 32*(motion_center[8:5])) &
        (ypos < (32*(motion_center[8:5])+32)) )
            //if (center_ready & (xpos >=
            // (xpos <
            // (ypos >=
            // (ypos <
                begin
                    //center block
                    vga_out_red <= 255;  // yellow block
                    vga_out_green <= 255;
                    vga_out_blue <= 0;
                end
        else if (center_ready & (xpos >= 32*(motion_mb1[4:0])) & (xpos <
        (32*(motion_mb1[4:0])+32)) &
        (ypos >= 32*(motion_mb1[8:5])) &
        (ypos < (32*(motion_mb1[8:5])+32)) )
            begin
                    //draw motion mb 1
                    vga_out_red <= 255;  // red motion block
                    vga_out_green <= 0;
                    vga_out_blue <= 0;
                end
        else if (center_ready & (xpos >= 32*(motion_mb2[4:0])) & (xpos <
        (32*(motion_mb2[4:0])+32)) &
        (ypos >= 32*(motion_mb2[8:5])) &
        (ypos < (32*(motion_mb2[8:5])+32)) )
            begin
                    //draw motion mb 2
                    vga_out_red <= 0;  // green motion block
                    vga_out_green <= 255;
                    vga_out_blue <= 0;
                end
        else if (center_ready & (xpos >= 32*(motion_mb3[4:0])) & (xpos <
        (32*(motion_mb3[4:0])+32)) &
        (ypos >= 32*(motion_mb3[8:5])) &
        (ypos < (32*(motion_mb3[8:5])+32)) )
            begin
                    //draw motion mb 3
                    vga_out_red <= 0;
                    vga_out_green <= 0;
                    vga_out_blue <= 255;  // blue motion block
                end
        else if (center_ready & (xpos >= 32*(motion_mb4[4:0])) & (xpos <
        (32*(motion_mb4[4:0])+32)) &
        (ypos >= 32*(motion_mb4[8:5])) & (ypos <
        (32*(motion_mb4[8:5])+32)) )
            begin
                //draw motion mb 4
                vga_out_red <= 0;
vga_out_green <= 128; // teal motion block
vga_out_blue <= 126;
else
begin
  vga_out_red <= pixdata; // pass data
  vga_out_green <= pixdata;
  vga_out_blue <= pixdata;
end
else
begin
  vga_out_red <= 255; // show white
  vga_out_green <= 255;
  vga_out_blue <= 255;
end
else
begin
  vga_out_red <= pixdata; // for black and white display -->
  vga_out_green <= pixdata;
  vga_out_blue <= pixdata;
end
// debug
// assign debug = {56'b0, difference_luma};
assign debug = debug_camera_angle;
endmodule
module ntsc_decode(clk, reset, tv_in_ycrb, ycrb, f, v, h, data_valid);

    // clk - line-locked clock (in this case, LLC1 which runs at 27Mhz)
    // reset - system reset
    // tv_in_ycrb - 10-bit input from chip. should map to pins [19:10]
    // ycrb - 24 bit luminance and chrominance (8 bits each)
    // f - field: 1 indicates an even field, 0 an odd field
    // v - vertical sync: 1 means vertical sync
    // h - horizontal sync: 1 means horizontal sync

    input clk;
    input reset;
    input [9:0] tv_in_ycrb; // modified for 10 bit input - should be P[19:10]
    output [29:0] ycrb;
    output f;
    output v;
    output h;
    output data_valid;
    // output [4:0] state;

    parameter SYNC_1 = 0;
    parameter SYNC_2 = 1;
    parameter SYNC_3 = 2;
    parameter SAV_f1_cb0 = 3;
    parameter SAV_f1_y0 = 4;
    parameter SAV_f1_cr1 = 5;
    parameter SAV_f1_y1 = 6;
    parameter EAV_f1 = 7;
    parameter SAV_VBI_f1 = 8;
    parameter EAV_VBI_f1 = 9;
    parameter SAV_f2_cb0 = 10;
    parameter SAV_f2_y0 = 11;
    parameter SAV_f2_cr1 = 12;
    parameter SAV_f2_y1 = 13;
    parameter EAV_f2 = 14;
    parameter SAV_VBI_f2 = 15;
    parameter EAV_VBI_f2 = 16;

    // In the start state, the module doesn't know where
    // in the sequence of pixels, it is looking.

    // Once we determine where to start, the FSM goes through a normal
    // sequence of SAV process_YCrCb EAV... repeat

    // The data stream looks as follows
/** SAV_FF | SAV_00 | SAV_00 | SAV_XY | Cb0 | Y0 | Cr1 | Y1 | Cb2 | Y2 | ... | EAV
sequence
/** There are two things we need to do:
/** 1. Find the two SAV blocks (stands for Start Active Video perhaps?)
/** 2. Decode the subsequent data

reg [4:0] current_state = 5'h00;
reg [9:0] y = 10'h000; // luminance
reg [9:0] cr = 10'h000; // chrominance
reg [9:0] cb = 10'h000; // more chrominance
assign state = current_state;

always @(posedge clk)
begin
  if (reset)
  begin
  end
  else
  begin
    // these states don't do much except allow us to know where we are in the stream.
    // whenever the synchronization code is seen, go back to the sync_state before
    // transitioning to the new state
    case (current_state)
      SYNC_1: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_2 : SYNC_1;
      SYNC_2: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_3 : SYNC_1;
      SYNC_3: current_state <= (tv_in_ycrcb == 10'h200) ? SAV_f1_cb0 :
                 (tv_in_ycrcb == 10'h274) ? EAV_f1 :
                 (tv_in_ycrcb == 10'h2ac) ? SAV_VBI_f1 :
                 (tv_in_ycrcb == 10'h2d8) ? EAV_VBI_f1 :
                 (tv_in_ycrcb == 10'h31c) ? SAV_f2_cb0 :
                 (tv_in_ycrcb == 10'h368) ? EAV_f2 :
                 (tv_in_ycrcb == 10'h3b0) ? SAV_VBI_f2 :
                 (tv_in_ycrcb == 10'h3c4) ? EAV_VBI_f2 : SYNC_1;
      SAV_f1_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y0;
      SAV_f1_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cr1;
      SAV_f1_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y1;
      SAV_f1_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cb0;
      SAV_f2_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_y0;
      SAV_f2_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_cr1;
      SAV_f2_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_y1;
      SAV_f2_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_cb0;

      // These states are here in the event that we want to cover these signals // in the future. For now, they just send the state machine back to SYNC_1
      EAV_f1: current_state <= SYNC_1;
      EAV_VBI_f1: current_state <= SYNC_1;
      EAV_VBI_f2: current_state <= SYNC_1;
      SAV_VBI_f1: current_state <= SYNC_1;
      SAV_VBI_f2: current_state <= SYNC_1;
      endcase
  end
end // always @(posedge clk)

// implement our decoding mechanism
wire y_enable;
wire cr_enable;
wire cb_enable;

// if y is coming in, enable the register // likewise for cr and cb
assign y_enable = (current_state == SAV_f1_y0) ||
                   (current_state == SAV_f1_y1) ||
(current_state == SAV_f2_y0) ||
(current_state == SAV_f2_y1);
assign cr_enable = (current_state == SAV_f1_cr1) ||
(current_state == SAV_f2_cr1);
assign cb_enable = (current_state == SAV_f1_cb0) ||
(current_state == SAV_f2_cb0);
// f, v, and h only go high when active
assign {v, h} = (current_state == SYNC_3) ? tv_in_ycrcb[7:6] : 2'b00;
// data is valid when we have all three values: y, cr, cb
assign data_valid = y_enable;
assign ycrcb = {y, cr, cb};
reg f = 0;
always @(posedge clk)
begin
  y <= y_enable ? tv_in_ycrcb : y;
  cr <= cr_enable ? tv_in_ycrcb : cr;
  cb <= cb_enable ? tv_in_ycrcb : cb;
end
endmodule

`define INPUT_SELECT 4'h0
// 0: CVBS on AIN1 (composite video in)
// 7: Y on AIN2, C on AIN5 (s-video in)
// (These are the only configurations supported by the 6.111 labkit hardware)
`define INPUT_MODE 4'h0
// 0: Autodetect: NTSC or PAL (BGHID), w/o pedestal
// 1: Autodetect: NTSC or PAL (BGHID), w/pedestal
// 2: Autodetect: NTSC or PAL (N), w/o pedestal
// 3: Autodetect: NTSC or PAL (N), w/pedestal
// 4: NTSC w/o pedestal
// 5: NTSC w/pedestal
// 6: NTSC 4.43 w/o pedestal
// 7: NTSC 4.43 w/pedestal
// 8: PAL BGHID w/o pedestal
// 9: PAL N w/pedestal
// A: PAL M w/o pedestal
// B: PAL M w/pedestal
// C: PAL combination N
// D: PAL combination N w/pedestal
// E-F: [Not valid]
`define ADV7185_REGISTER_0 `{INPUT_MODE, `INPUT_SELECT}
`define VIDEO_QUALITY 2'h0
// 0: Broadcast quality
// 1: TV quality
// 2: VCR quality
// 3: Surveillance quality
`define SQUARE_PIXEL_IN_MODE 1'b0
// 0: Normal mode
// 1: Square pixel mode
`define DIFFERENTIAL_INPUT 1'b0
// 0: Single-ended inputs
// 1: Differential inputs
`define FOUR_TIMES_SAMPLING 1'b0
// 0: Standard sampling rate
// 1: 4x sampling rate (NTSC only)
`define BETACAM 1'b0
// 0: Standard video input
// 1: Betacam video input
`define AUTOMATIC_STARTUP_ENABLE 1'b1
// 0: Change of input triggers reacquire
// 1: Change of input does not trigger reacquire
`define ADV7185_REGISTER_1 {`AUTOMATIC_STARTUP_ENABLE, 1'b0, `BETACAM, `FOUR_TIMES_SAMPLING, `DIFFERENTIAL_INPUT, `SQUARE_PIXEL_IN_MODE, `VIDEO_QUALITY}

// 0: Composite =  4.5dB, s-video =  9.25dB
// 1: Composite =  4.5dB, s-video =  9.25dB
// 2: Composite =  4.5dB, s-video =  5.75dB
// 3: Composite =  1.25dB, s-video =  3.3dB
// 4: Composite =  0.0dB, s-video =  0.0dB
// 5: Composite = -1.25dB, s-video = -3.0dB
// 6: Composite = -1.75dB, s-video = -8.0dB
// 7: Composite = -3.0dB, s-video = -8.0dB
`define Y_PEAKING_FILTER 3'h4
// 0: No coring
// 1: Truncate if Y < black+8
// 2: Truncate if Y < black+16
// 3: Truncate if Y < black+32
`define ADV7185_REGISTER_2 {3'b000, `CORING, `Y_PEAKING_FILTER}

// 0: Philips-compatible
// 1: Broktree API A-compatible
// 2: Broktree API B-compatible
// 3: [Not valid]
`define INTERFACE_SELECT 2'h0
`define OUTPUT_FORMAT 4'h0
// 0: 10-bit @ LLC, 4:2:2 CCIR656
// 1: 20-bit @ LLC, 4:2:2 CCIR656
// 2: 16-bit @ LLC, 4:2:2 CCIR656
// 3: 8-bit @ LLC, 4:2:2 CCIR656
// 4: 12-bit @ LLC, 4:1:1
// 5-F: [Not valid]
// (Note that the 6.111 labkit hardware provides only a 10-bit interface to
// the ADV7185.)
`define TRISTATE_OUTPUT_DRIVERS 1'b0
// 0: Drivers Tristated when -OE is high
// 1: Drivers always tristated
`define VBI_ENABLE 1'b0
// 0: Decode lines during vertical blanking interval
// 1: Decode only active video regions
`define ADV7185_REGISTER_3 {`VBI_ENABLE, `TRISTATE_OUTPUT_DRIVERS, `OUTPUT_FORMAT, `INTERFACE_SELECT}
Register 4

`define OUTPUT_DATA_RANGE 1'b0
// 0: Output values restricted to CCIR-compliant range
// 1: Use full output range
`define BT656_TYPE 1'b0
// 0: BT656-3-compatible
// 1: BT656-4-compatible
`define ADV7185_REGISTER_4 (`BT656_TYPE, 3'b000, 3'b110, `OUTPUT_DATA_RANGE)

Register 5

`define GENERAL_PURPOSE_OUTPUTS 4'b0000
`define GPO_0_1_ENABLE 1'b0
// 0: General purpose outputs 0 and 1 tristated
// 1: General purpose outputs 0 and 1 enabled
`define GPO_2_3_ENABLE 1'b0
// 0: General purpose outputs 2 and 3 tristated
// 1: General purpose outputs 2 and 3 enabled
`define BLANK_CHROMA_IN_VBI 1'b1
// 0: Chroma decoded and output during vertical blanking
// 1: Chroma blanked during vertical blanking
`define HLOCK_ENABLE 1'b0
// 0: GPO 0 is a general purpose output
// 1: GPO 0 shows HLOCK status
`define ADV7185REGISTER_5 (`HLOCK_ENABLE, `BLANK_CHROMA_IN_VBI, `GPO_2_3_ENABLE,
`GPO_0_1_ENABLE, `GENERAL_PURPOSE_OUTPUTS)

Register 7

`define FIFO_FLAG_MARGIN 5'h10
// Sets the locations where FIFO almost-full and almost-empty flags are set
`define FIFO_RESET 1'b0
// 0: Normal operation
// 1: Reset FIFO. This bit is automatically cleared
`define AUTOMATIC_FIFO_RESET 1'b0
// 0: No automatic reset
// 1: FIFO is automatically reset at the end of each video field
`define FIFO_FLAG_SELF_TIME 1'b1
// 0: FIFO flags are synchronized to CLINK
// 1: FIFO flags are synchronized to internal 27MHz clock
`define ADV7185_REGISTER_7 (`FIFO_FLAG_SELF_TIME, `AUTOMATIC_FIFO_RESET, `FIFO_RESET,
`FIFO_FLAG_MARGIN)

Register 8

`define INPUT_CONTRAST_ADJUST 8'h80
`define ADV7185_REGISTER_8 (`INPUT_CONTRAST_ADJUST)

Register 9

`define INPUT_SATURATION_ADJUST 8'h8C
`define ADV7185_REGISTER_9 (`INPUT_SATURATION_ADJUST)

Register A
`define INPUT_BRIGHTNESS_ADJUST 8'h00
`define ADV7185_REGISTER_A {`INPUT_BRIGHTNESS_ADJUST}

// Register B

`define INPUT_HUE_ADJUST 8'h00
`define ADV7185_REGISTER_B {`INPUT_HUE_ADJUST}

// Register C

`define DEFAULT_VALUE_ENABLE 1'b0
// 0: Use programmed Y, Cr, and Cb values
// 1: Use default values
`define DEFAULT_VALUE_AUTOMATIC_ENABLE 1'b0
// 0: Use programmed Y, Cr, and Cb values
// 1: Use default values if lock is lost
`define DEFAULT_Y_VALUE 6'h0C
// Default Y value

`define ADV7185_REGISTER_C {1'b0, `DEFAULT_Y_VALUE, 1'b0, `DEFAULT_VALUE_AUTOMATIC_ENABLE, `DEFAULT_VALUE_ENABLE}

// Register D

`define DEFAULT_CR_VALUE 4'h8
// Most-significant four bits of default Cr value
`define DEFAULT_CB_VALUE 4'h8
// Most-significant four bits of default Cb value

`define ADV7185_REGISTER_D {4'h8, 4'h8}

// Register E

`define TEMPORAL_DECIMATION_ENABLE 1'b0
// 0: Disable
// 1: Enable
`define TEMPORAL_DECIMATION_CONTROL 2'h0
// 0: Supress frames, start with even field
// 1: Supress frames, start with odd field
// 2: Supress even fields only
// 3: Supress odd fields only
`define TEMPORAL_DECIMATION_RATE 4'h0
// 0-F: Number of fields/frames to skip

`define ADV7185_REGISTER_E {1'b0, `TEMPORAL_DECIMATION_RATE, `TEMPORAL_DECIMATION_CONTROL, 1'b0}

// Register F

`define POWER_SAVE_CONTROL 2'h0
// 0: Full operation
// 1: CVBS only
// 2: Digital only
// 3: Power save mode
`define POWER_DOWN_SOURCE_PRIORITY 1'b0
// 0: Power-down pin has priority
// 1: Power-down control bit has priority
`define POWER_DOWN_REFERENCE  1'b0  // 0: Reference is functional
       // 1: Reference is powered down
`define POWER_DOWN_LLC_GENERATOR  1'b0 // 0: LLC generator is functional
       // 1: LLC generator is powered down
`define POWER_DOWN_CHIP  1'b0 // 0: Chip is functional
       // 1: Input pads disabled and clocks stopped
`define TIMING_REACQUIRE  1'b0 // 0: Normal operation
       // 1: Reacquire video signal (bit will automatically reset)
`define RESET_CHIP  1'b0 // 0: Normal operation
       // 1: Reset digital core and I2C interface (bit will automatically reset)

`define ADV7185_REGISTER_F {`RESET_CHIP, `TIMING_REACQUIRE, `POWER_DOWN_CHIP,
       `POWER_DOWN_LLC_GENERATOR, `POWER_DOWN_REFERENCE, `POWER_DOWN_SOURCE_PRIORITY,
       `POWER_SAVE_CONTROL}

`define PEAK_WHITE_UPDATE  1'b1 // 0: Update gain once per line
       // 1: Update gain once per frame
`define AVERAGE_BRIGHTNESS_LINES  1'b1 // 0: Use lines 33 to 310
       // 1: Use lines 33 to 270
`define MAXIMUM_IRES  3'h0 // 0: PAL: 133, NTSC: 122
       // 1: PAL: 125, NTSC: 115
       // 2: PAL: 120, NTSC: 110
       // 3: PAL: 115, NTSC: 105
       // 4: PAL: 110, NTSC: 100
       // 5: PAL: 105, NTSC: 100
       // 6-7: PAL: 100, NTSC: 100
`define COLOR_KILL  1'b1 // 0: Disable color kill
       // 1: Enable color kill

`define ADV7185_REGISTER_33 {1'b1, `COLOR_KILL, 1'b1, `MAXIMUM_IRES,
       `AVERAGE_BRIGHTNESS_LINES, `PEAK_WHITE_UPDATE}

#define ADV7185_REGISTER_10 8'h00
#define ADV7185_REGISTER_11 8'h00
#define ADV7185_REGISTER_12 8'h00
#define ADV7185_REGISTER_13 8'h45
#define ADV7185_REGISTER_14 8'h18
#define ADV7185_REGISTER_15 8'h60
#define ADV7185_REGISTER_16 8'h00
#define ADV7185_REGISTER_17 8'h01
#define ADV7185_REGISTER_18 8'h00
#define ADV7185_REGISTER_19 8'h10
#define ADV7185_REGISTER_1A 8'h10
#define ADV7185_REGISTER_1B 8'hF0
#define ADV7185_REGISTER_1C 8'h16
#define ADV7185_REGISTER_1D 8'h01
#define ADV7185_REGISTER_1E 8'h00
#define ADV7185_REGISTER_1F 8'h3D
#define ADV7185_REGISTER_20 8'hD0
#define ADV7185_REGISTER_21 8'h09
#define ADV7185_REGISTER_22 8'h8C
#define ADV7185_REGISTER_23 8'hE2
#define ADV7185_REGISTER_24 8'h1F
#define ADV7185_REGISTER_25 8'h07
#define ADV7185_REGISTER_26 8'hC2
#define ADV7185_REGISTER_27 8'h58
#define ADV7185_REGISTER_28 8'h3C
#define ADV7185_REGISTER_29 8'h00
module adv7185init (reset, clock_27mhz, source, tv_in_reset_b, tv_in_i2c_clock, tv_in_i2c_data);

input reset;
input clock_27mhz;
output tv_in_reset_b; // Reset signal to ADV7185
output tv_in_i2c_clock; // I2C clock output to ADV7185
output tv_in_i2c_data; // I2C data line to ADV7185
input source; // 0: composite, 1: s-video

initial begin
$display("ADV7185 Initialization values:");
$display(" Register 0: 0x%X", ADV7185_REGISTER_0);
$display(" Register 1: 0x%X", ADV7185_REGISTER_1);
$display(" Register 2: 0x%X", ADV7185_REGISTER_2);
$display(" Register 3: 0x%X", ADV7185_REGISTER_3);
$display(" Register 4: 0x%X", ADV7185_REGISTER_4);
$display(" Register 5: 0x%X", ADV7185_REGISTER_5);
$display(" Register 7: 0x%X", ADV7185_REGISTER_7);
$display(" Register 8: 0x%X", ADV7185_REGISTER_8);
$display(" Register 9: 0x%X", ADV7185_REGISTER_9);
$display(" Register A: 0x%X", ADV7185_REGISTER_A);
$display(" Register B: 0x%X", ADV7185_REGISTER_B);
$display(" Register C: 0x%X", ADV7185_REGISTER_C);
$display(" Register D: 0x%X", ADV7185_REGISTER_D);
$display(" Register E: 0x%X", ADV7185_REGISTER_E);
$display(" Register F: 0x%X", ADV7185_REGISTER_F);
$display(" Register 33: 0x%X", ADV7185_REGISTER_33);
end

// Generate a 1MHz for the I2C driver (resulting I2C clock rate is 250kHz)

reg [7:0] clk_div_count, reset_count;
reg clock_slow;
wire reset_slow;

initial begin
clk_div_count <= 8'h00;
// synthesis attribute init of clk_div_count is "00"

clock_slow <= 1'b0;
// synthesis attribute init of clock_slow is "0"
end
always @(posedge clock_27mhz)
  if (clk_div_count == 26)
    begin
      clock_slow <= ~clock_slow;
      clk_div_count <= 0;
    end
  else
    clk_div_count <= clk_div_count+1;
always @(posedge clock_27mhz)
  if (reset)
    reset_count <= 100;
  else
    reset_count <= (reset_count==0) ? 0 : reset_count-1;
assign reset_slow = reset_count != 0;

// I2C driver
//
reg load;
reg [7:0] data;
wire ack, idle;
i2c i2c(.reset(reset_slow), .clock4x(clock_slow), .data(data), .load(load),
    .ack(ack), .idle(idle), .scl(tv_in_i2c_clock),
    .sda(tv_in_i2c_data));

// State machine
//
reg [7:0] state;
reg tv_in_reset_b;
reg old_source;
always @(posedge clock_slow)
  if (reset_slow)
    begin
      state <= 0;
      load <= 0;
      tv_in_reset_b <= 0;
      old_source <= 0;
    end
  else
    case (state)
      8'h00:
        begin
          // Assert reset
          load <= 1'b0;
          tv_in_reset_b <= 1'b0;
          if (!ack)
            state <= state+1;
        end
      8'h01:
        state <= state+1;
      8'h02:
        begin
          // Release reset
          tv_in_reset_b <= 1'b1;
          state <= state+1;
        end
      8'h03:
        begin
          // Send ADV7185 address
          data <= 8'h8A;
          load <= 1'b1;
          if (ack)
            state <= state+1;
        end
    endcase;
8'h04:
begin
  // Send subaddress of first register
  data <= 8'h00;
  if (ack)
    state <= state+1;
end
8'h05:
begin
  // Write to register 0
  data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
  if (ack)
    state <= state+1;
end
8'h06:
begin
  // Write to register 1
  data <= `ADV7185_REGISTER_1;
  if (ack)
    state <= state+1;
end
8'h07:
begin
  // Write to register 2
  data <= `ADV7185_REGISTER_2;
  if (ack)
    state <= state+1;
end
8'h08:
begin
  // Write to register 3
  data <= `ADV7185_REGISTER_3;
  if (ack)
    state <= state+1;
end
8'h09:
begin
  // Write to register 4
  data <= `ADV7185_REGISTER_4;
  if (ack)
    state <= state+1;
end
8'h0A:
begin
  // Write to register 5
  data <= `ADV7185_REGISTER_5;
  if (ack)
    state <= state+1;
end
8'h0B:
begin
  // Write to register 6
  data <= 8'h00; // Reserved register, write all zeros
  if (ack)
    state <= state+1;
end
8'h0C:
begin
  // Write to register 7
  data <= `ADV7185_REGISTER_7;
  if (ack)
    state <= state+1;
end
8'h0D:
begin
  // Write to register 8
  data <= `ADV7185_REGISTER_8;
  if (ack)
    state <= state+1;
end
8'h0E:
begin
  // Write to register 9
  data <= `ADV7185_REGISTER_9;
  if (ack)
    state <= state+1;
end
8'h0F: begin
  // Write to register A
  data <= `ADV7185_REGISTER_A;
  if (ack)
    state <= state+1;
end
8'h10: begin
  // Write to register B
  data <= `ADV7185_REGISTER_B;
  if (ack)
    state <= state+1;
end
8'h11: begin
  // Write to register C
  data <= `ADV7185_REGISTER_C;
  if (ack)
    state <= state+1;
end
8'h12: begin
  // Write to register D
  data <= `ADV7185_REGISTER_D;
  if (ack)
    state <= state+1;
end
8'h13: begin
  // Write to register E
  data <= `ADV7185_REGISTER_E;
  if (ack)
    state <= state+1;
end
8'h14: begin
  // Write to register F
  data <= `ADV7185_REGISTER_F;
  if (ack)
    state <= state+1;
end
8'h15: begin
  // Wait for I2C transmitter to finish
  load <= 1'b0;
  if (idle)
    state <= state+1;
end
8'h16: begin
  // Write address
  data <= 8'h8A;
  load <= 1'b1;
  if (ack)
    state <= state+1;
end
8'h17: begin
  data <= 8'h33;
  if (ack)
    state <= state+1;
end
8'h18: begin
  data <= `ADV7185_REGISTER_33;
if (ack)
    state <= state+1;
end
8'h19: begin
    load <= 1'b0;
    if (idle)
        state <= state+1;
end
8'h1A: begin
    data <= 8'h8A;
    load <= 1'b1;
    if (ack)
        state <= state+1;
end
8'h1B: begin
    data <= 8'h33;
    if (ack)
        state <= state+1;
end
8'h1C: begin
    load <= 1'b0;
    if (idle)
        state <= state+1;
end
8'h1D: begin
    load <= 1'b1;
    data <= 8'h8B;
    if (ack)
        state <= state+1;
end
8'h1E: begin
    data <= 8'hFF;
    if (ack)
        state <= state+1;
end
8'h20: begin
    // Idle
    if (old_source != source) state <= state+1;
    old_source <= source;
end
8'h21: begin
    // Send ADV7185 address
    data <= 8'h8A;
    load <= 1'b1;
    if (ack) state <= state+1;
end
8'h22: begin
    // Send subaddress of register 0
    data <= 8'h00;
    if (ack) state <= state+1;
end
8'h23: begin
    // Write to register 0
    data <= `ADV7185REGISTER_0 | {3'h00, {3{source}}};
    if (ack) state <= state+1;
end
8'h24: begin
    // Wait for I2C transmitter to finish
module i2c (reset, clock4x, data, load, idle, ack, scl, sda);

input reset;
input clock4x;
input [7:0] data;
input load;
output idle;
output ack;
output scl;
output sda;

reg [7:0] ldata;
reg ack, idle;
reg scl;
reg sdai;
reg [7:0] state;

assign sda = sdai ? 1'bZ : 1'b0;

always @(posedge clock4x)
begin
  if (reset)
    begin
      state <= 0;
      ack <= 0;
    end
  else
    case (state)
      8'h00: // idle
        begin
          scl <= 1'b1;
          sdai <= 1'b1;
          ack <= 1'b0;
          idle <= 1'b1;
          if (load)
            begin
              ldata <= data;
              ack <= 1'b1;
              state <= state+1;
            end
        end
      8'h01: // Start
        begin
          ack <= 1'b0;
          idle <= 1'b0;
          sdai <= 1'b0;
          state <= state+1;
        end
      8'h02:
        begin
          scl <= 1'b0;
          state <= state+1;
        end
      8'h03: // Send bit 7
        begin
          ack <= 1'b0;
          sdai <= ldata[7];
          state <= state+1;
        end
      8'h04:
        begin
          scl <= 1'b1;
        end
    endcase
endmodule
state <= state+1;
end
8'h05:
begin
state <= state+1;
end
8'h06:
begin
scl <= 1'b0;
state <= state+1;
end
8'h07:
begin
sdai <= ldata[6];
state <= state+1;
end
8'h08:
begin
scl <= 1'b1;
state <= state+1;
end
8'h09:
begin
state <= state+1;
end
8'h0A:
begin
scl <= 1'b0;
state <= state+1;
end
8'h0B:
begin
sdai <= ldata[5];
state <= state+1;
end
8'h0C:
begin
scl <= 1'b1;
state <= state+1;
end
8'h0D:
begin
state <= state+1;
end
8'h0E:
begin
scl <= 1'b0;
state <= state+1;
end
8'h0F:
begin
sdai <= ldata[4];
state <= state+1;
end
8'h10:
begin
scl <= 1'b1;
state <= state+1;
end
8'h11:
begin
state <= state+1;
end
8'h12:
begin
scl <= 1'b0;
state <= state+1;
end
8'h13:
begin
sdai <= ldata[3];
state <= state + 1;
end
8'h14:
begin
  scl <= 1'b1;
  state <= state + 1;
end
8'h15:
begin
  state <= state + 1;
end
8'h16:
begin
  scl <= 1'b0;
  state <= state + 1;
end
8'h17:
begin
  sda1 <= ldata[2];
  state <= state + 1;
end
8'h18:
begin
  scl <= 1'b1;
  state <= state + 1;
end
8'h19:
begin
  state <= state + 1;
end
8'h1A:
begin
  scl <= 1'b0;
  state <= state + 1;
end
8'h1B:
begin
  sda1 <= ldata[1];
  state <= state + 1;
end
8'h1C:
begin
  scl <= 1'b1;
  state <= state + 1;
end
8'h1D:
begin
  state <= state + 1;
end
8'h1E:
begin
  scl <= 1'b0;
  state <= state + 1;
end
8'h1F:
begin
  sda1 <= ldata[0];
  state <= state + 1;
end
8'h20:
begin
  scl <= 1'b1;
  state <= state + 1;
end
8'h21:
begin
  state <= state + 1;
end
8'h22:
begin
  scl <= 1'b0;
state <= state+1;
end
8'h23: // Acknowledge bit
begin
    state <= state+1;
end
8'h24:
begin
    scl <= 1'b1;
    state <= state+1;
end
8'h25:
begin
    state <= state+1;
end
8'h26:
begin
    scl <= 1'b0;
    if (load)
        begin
            ldata <= data;
            ack <= 1'b1;
            state <= 3;
        end
    else
        state <= state+1;
end
8'h27:
begin
    sdai <= 1'b0;
    state <= state+1;
end
8'h28:
begin
    scl <= 1'b1;
    state <= state+1;
end
8'h29:
begin
    sdai <= 1'b1;
    state <= 0;
end
endcase
endmodule
module videoram(
    addra,
    addrb,
    clka,
    clkb,
    dinb,
    douta,
    doutb,
    web);

input [15 : 0] addra;
input [15 : 0] addrb;
input clka;
input clkb;
input [7 : 0] dinb;
output [7 : 0] douta;
output [7 : 0] doutb;
input web;

// synopsys translate_off
BLKMEMDP_V6_1 #(
    16, // c_addra_width
    16, // c_addrb_width
    "0", // c_default_data
    49152, // c_depth_a
    49152, // c_depth_b
    0, // c_enable_rlocs
    1, // c_has_default_data
    0, // c_has_dina
    1, // c_has_dinb
    1
)
// synopsys translate_on
inst (.ADDRA(addra),
  .ADDRB(addrb),
  .CLKA(clka),
  .CLKB(clkb),
  .DINB(dinb),
  .DOUTA(douta),
  .DOUTB(doutb),
  .WEB(web),
  .DINA(),
  .ENA(),
  .ENB(),
  .NDA(),
  .NDB(),
  .RFDA(),
  .RFDB(),
  .RDYA(),
  .RDYB(),
  .SINITA(),
  .SINITB(),
  .WEA());

// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of videoram is "true"
// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of videoram is "black_box"
endmodule
// generate display pixels from reading the ZBT ram
// note that the ZBT ram has 2 cycles of read (and write) latency
// We take care of that by latching the data at an appropriate time.
// Note that the ZBT stores 36 bits per word; we use only 32 bits here,
// decoded into four bytes of pixel data.

module vram_display(reset,clk,hcount,vcount,vr_pixel,
    vram_addr,vram_read_data);

    input reset, clk;
    input [10:0] hcount;
    input [9:0] vcount;
    output [7:0] vr_pixel;
    output [18:0] vram_addr;
    input [35:0] vram_read_data;

    wire [18:0] vram_addr = {1'b0, vcount, hcount[9:2]};
    wire [1:0] hc4 = hcount[1:0];
    reg [7:0] vr_pixel;
    reg [35:0] vr_data_latched;
    reg [35:0] last_vr_data;

    always @(posedge clk)
        last_vr_data <= (hc4==2'd3) ? vr_data_latched : last_vr_data;
    always @(posedge clk)
        vr_data_latched <= (hc4==2'd1) ? vram_read_data : vr_data_latched;
    always @(*) // each 36-bit word from RAM is decoded to 4 bytes
        case (hc4)
            2'd3: vr_pixel = last_vr_data[7:0];
            2'd2: vr_pixel = last_vr_data[7+8:0+8];
            2'd1: vr_pixel = last_vr_data[7+16:0+16];
            2'd0: vr_pixel = last_vr_data[7+24:0+24];
        endcase
endmodule // vram_display
module xvga(vclock,hcount,vcount,hsync,vsync,blank);
    input vclock;
    output [10:0] hcount;
    output [9:0] vcount;
    output vsync;
    output hsync;
    output blank;

    reg hsync,vsync,hblank,vblank,blank;
    reg [10:0] hcount; // pixel number on current line
    reg [9:0] vcount; // line number

    // horizontal: 1344 pixels total
    // display 1024 pixels per line
    wire hsyncon,hsyncoff,hreset,hblankon;
    assign hblankon = (hcount == 1023);
    assign hsyncon = (hcount == 1047);
    assign hsyncoff = (hcount == 1183);
    assign hreset = (hcount == 1343);

    // vertical: 806 lines total
    // display 768 lines
    wire vsyncon,vsyncoff,vreset,vblankon;
    assign vblankon = hreset & (vcount == 767);
    assign vsyncon = hreset & (vcount == 776);
    assign vsyncoff = hreset & (vcount == 782);
    assign vreset = hreset & (vcount == 805);

    // sync and blanking
    wire next_hblank,next_vblank;
    assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
    assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;

    always @(posedge vclock) begin
        hcount <= hreset ? 0 : hcount + 1;
        hblank <= next_hblank;
        hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync; // active low

        vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
        vblank <= next_vblank;
        vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync; // active low

        blank <= next_vblank | (next_hblank & ~hreset);
    end
endmodule
module zbt_6111(clk, cen, we, addr, write_data, read_data, ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b);

input clk;   // system clock
input cen;   // clock enable for gating ZBT cycles
input we;   // write enable (active HIGH)
input [18:0] addr;  // memory address
input [35:0] write_data; // data to write
output [35:0] read_data; // data read from memory
output ram_clk; // physical line to ram clock
output ram_we_b; // physical line to ram we_b
output [18:0] ram_address; // physical line to ram address
output ram_data; // physical line to ram data
output ram_cen_b; // physical line to ram clock enable

// clock enable (should be synchronous and one cycle high at a time)
wire ram_cen_b = ~cen;

// create delayed ram_we signal: note the delay is by two cycles!
// ie we present the data to be written two cycles after we is raised
// this means the bus is tri-stated two cycles after we is raised.
reg [1:0] we_delay;

always @(posedge clk)
we_delay <= cen ? {we_delay[0],we} : we_delay;

// create two-stage pipeline for write data
reg [35:0] write_data_old1;
reg [35:0] write_data_old2;
always @(posedge clk)
if (cen)
{write_data_old2, write_data_old1} <= {write_data_old1, write_data};

// wire to ZBT RAM signals
assign ram_we_b = ~we;
assign ram_clk = ~clk; // RAM is not happy with our data hold times if its clk edges equal FPGA's
// so we clock it on the falling edges // and thus let data stabilize longer
assign ram_address = addr;
assign ram_data = we_delay[1] ? write_data_old2 : {36{1'bZ}};
assign read_data = ram_data;
endmodule // zbt_6111
/*******************************/
// parameterized delay line
module delayN(clk,in,out);
  input clk;
  input in;
  output out;
  parameter NDELAY = 3;
  reg [NDELAY-1:0] shiftreg;
  wire out = shiftreg[NDELAY-1];
  always @(posedge clk)
    shiftreg <= {shiftreg[NDELAY-2:0],in};
endmodule // delayN
module ntsc_to_zbt(clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data, ntsc_we, sw);

input   clk; // system clock
input   vclk; // video clock from camera
input [2:0]   fvh;
input [7:0]   din;
output [18:0] ntsc_addr;
output [35:0] ntsc_data;
output   ntsc_we; // write enable for NTSC data
input   sw;  // switch which determines mode (for debugging)

parameter   COL_START = 10'd30;
parameter   ROW_START = 10'd30;

// here put the luminance data from the ntsc decoder into the ram
// this is for 1024 x 768 XGA display
reg [9:0]   col = 0;
reg [9:0]   row = 0;
reg [7:0]   vdata = 0;
reg    vwe;
reg    old_dv;
reg    old_frame; // frames are even / odd interlaced
reg    even_odd; // decode interlaced frame to this wire
wire   frame = fvh[2];
wire   frame_edge = frame & ~old_frame;
always @(posedge vclk) //LLC1 is reference
begin
    old_dv <= dv;
vwe <= dv && !fvh[2] && ~old_dv; // if data valid, write it
old_frame <= frame;
even_odd = frame_edge ? ~even_odd : even_odd;
if (!fvh[2])
begin
    col <= fvh[0] ? COL_START :
    row <= fvh[1] ? ROW_START :
vdata <= (dv && !fvh[2]) ? din : vdata;
end
end

// synchronize with system clock
reg [9:0] x[1:0], y[1:0];
reg [7:0] data[1:0];
reg    we[1:0];
reg    eo[1:0];
always @(posedge clk)
begin

```
\{x[1], x[0]\} <= \{x[0], col\};
\{y[1], y[0]\} <= \{y[0], row\};
\{data[1], data[0]\} <= \{data[0], vdata\};
\{we[1], we[0]\} <= \{we[0], vwe\};
\{eo[1], eo[0]\} <= \{eo[0], even_odd\};
end

// edge detection on write enable signal
reg old_we;
wire we_edge = we[1] & ~old_we;
always @(posedge clk) old_we <= we[1];

// shift each set of four bytes into a large register for the ZBT
reg [31:0] mydata;
always @(posedge clk)
if (we_edge)
  mydata <= \{ mydata[23:0], data[1] \};

// compute address to store data in
wire [18:0] myaddr = \{1'b0, y[1][8:0], eo[1], x[1][9:2]\};

// alternate (256x192) image data and address
wire [31:0] mydata2 = \{data[1], data[1], data[1], data[1]\};
wire [18:0] myaddr2 = \{1'b0, y[1][8:0], eo[1], x[1][7:0]\};

// update the output address and data only when four bytes ready
reg [18:0] ntsc_addr;
reg [35:0] ntsc_data;
wire ntsc_we = sw ? we_edge : (we_edge & (x[1][1:0] == 2'b00));
always @(posedge clk)
if (ntsc_we)
begin
  ntsc_addr <= sw ? myaddr : myaddr2; // normal and expanded modes
  ntsc_data <= sw ? \{4'b0, mydata2\} : \{4'b0, mydata\};
end
endmodule // ntsc_to_zbt